INSIDE MACINTOSH

PowerPC Numerics

Addison-Wesley Publishing Company

Reading, Massachusetts Menlo Park, California New York
Don Mills, Ontario Wokingham, England Amsterdam Bonn
Sydney Singapore Tokyo Madrid San Juan
Paris Seoul Milan Mexico City Taipei

Apple Computer, Inc.

No licenses, express or implied, are granted with respect to any of the technology described in this book. Apple retains all intellectual property rights associated with the technology described in this book. This book is intended to assist application developers to develop applications only for Apple Macintosh computers.

Every effort has been made to ensure that the information in this manual is accurate. Apple is not responsible for printing or clerical errors.

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

LIMITED WARRANTY ON MEDIA AND REPLACEMENT

ALL IMPLIED WARRANTIES ON THIS MANUAL, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has reviewed this manual, APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS OR IMPLIED, WITH RESPECT TO THIS MANUAL, ITS QUALITY, ACCURACY, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS MANUAL IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS MANUAL, even if advised of the possibility of such damages.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

ISBN 0-201-40728-0
1 2 3 4 5 6 7 8 9-CRW-9897969594
First Printing, March 1994

Library of Congress Cataloging-in-Publication Data

Book_title / [Apple Computer, Inc.].
p. cm.
Includes index.
ISBN 0-201-nnnnn-n
1. Macintosh (Computer)—Programming. 2.
I. Apple Computer, Inc.
nnnn.n.nnnnnnn 1994
nnn.nnn—nnnn 94-nnnnn
CIP

Contents

Chapter 3 Expression Evaluation 3-1

Chapter 4 Environmental Controls 4-1

Chapter 5 Conversions 5-1

Chapter 12 Assembly-Language Environmental Controls 12-1

Chapter 13 Assembly-Language Numeric Conversions 13-1

Chapter 14 Assembly-Language Numeric Operations 14-1

Appendix A SANE Versus PowerPC Numerics A-1

Appendix D FPCE Recommendations for Compilers D-1

Appendix E MathLib Reference E-1

Appendix F PowerPC Assembly-Language Numerics Reference F-1

Glossary GL-1

Bibliography BI-1

Index IN-1

Figures, Tables, and Listings

Chapter 1 IEEE Standard Arithmetic 1-1

Chapter 2 Floating-Point Data Formats 2-1

Chapter 3 Expression Evaluation 3-1

Chapter 4

Environmental Controls 4-1

Chapter 5

Conversions 5-1

Chapter 6

Numeric Operations and Functions 6-1

Chapter 7

Numeric Data Types in C 7-1

Chapter 8

Environmental Control Functions 8-1

Chapter 9

Conversion Functions 9-1

Chapter 11

Introduction to Assembly-Language Numerics 11-1

Chapter 12

Assembly-Language Environmental Controls 12-1

Chapter 13

Assembly-Language Numeric Conversions 13-1

Appendix A

SANE Versus PowerPC Numerics A-1

Appendix E

MathLib Reference E-1

PREFACE

About This Book

This book, *Inside Macintosh: PowerPC Numerics*, is the reference for the PowerPC Numerics environment. PowerPC Numerics is an environment in which floating-point operations are performed quickly and as accurately as possible. The PowerPC Numerics environment applies to Macintosh computers that use the PowerPC processor. The core features of PowerPC Numerics are not exclusive to Apple Computer; rather they are taken from IEEE Standard 754 for binary floating-point arithmetic and the standard proposed by the Floating-Point C Extensions (FPCE) branch of the Numerical C Extensions Group (ANSI X3J11.1).

In one sense, PowerPC Numerics is an abstraction: a definition of an environment for computer numerics, independent of a specific computer. To have an instance of this environment, you need a language in which to describe operations and an implementation unit to carry them out. The first part of this book describes the PowerPC Numerics definition, and the remaining parts describe how numerics is implemented in the PowerPC hardware and software.

You should read this book if

- you want to create PowerPC applications that use floating-point operations
- you have created a 680x0 application that uses floating-point operations and you plan to port it to PowerPC processor-based Macintosh computers (in this case, you might want to read Appendix A, “SANE Versus PowerPC Numerics,” first)
- you have not yet created a floating-point application, but you want to learn more about IEEE Standard 754 for binary floating-point arithmetic

This book is *not* for you if you don’t plan to port your 680x0 applications to the PowerPC environment. Applications that are 680x0 based will run on PowerPC processor-based Macintosh computers without rebuilding, but they use the Standard Apple Numerics Environment (SANE) in emulation instead of PowerPC Numerics. You should refer to the *Apple Numerics Manual*, second edition, which describes SANE.

Before reading this book, you should already be familiar with the PowerPC run-time architecture as described in *Inside Macintosh: PowerPC System Software.*

What’s in This Book

Part 1 describes the features shared by all PowerPC Numerics implementations and includes examples that show how to use PowerPC Numerics effectively. These examples are written in C, although other high-level languages might provide support for PowerPC Numerics. Read Part 1 to find out how PowerPC Numerics implements IEEE Standard 754 in general or to learn more about this standard.

Part 2 explains the numeric implementation in compilers and in the PowerPC Numerics library MathLib. This library is provided in ROM to implement both IEEE Standard 754 and the recommendations in the FPCE technical report. Part 2 is for use exclusively by C language programmers.

Part 3 explains the implementation in PowerPC hardware and the available assembly-language tools that perform numeric operations. Part 3 is for use by assembly-language programmers and by those who wish to look at compiler output.

The appendixes provide supplementary reference material. They give the differences between PowerPC Numerics and SANE, show how to port numerical programs to PowerPC processor-based Macintosh computers, provide listings of the header files in MathLib, and describe the FPCE recommendations for compilers. There are also summaries of the MathLib functions and PowerPC assembly-language floating-point instructions for your reference.

The bibliography at the end of this book lists some of the major sources on numerics. Refer to this bibliography for more extensive information on IEEE Standard 754, the FPCE technical report, or numerical programming in general. Also at the end of this book are a glossary of terms and an index.

Conventions Used in This Book

Inside Macintosh uses various conventions to present information. Words that require special treatment appear in specific fonts or font styles. Certain information appears in special formats so that you can scan it quickly.

Special Fonts

All code listings, reserved words, and the names of actual data structures, constants, fields, parameters, and routines are shown in Courier (`this is Courier`).

Words that appear in **boldface** are key terms or concepts and are defined in the glossary at the end of this book.

When a word or character appears in italics, it represents a variable that is replaced with a literal value in an actual computation. For example,

sqrt (*x*)

means take the square root of any floating-point value *x*, such as 1.45 or 2.789.

When a character appears in italics in one of the tables for special cases in Chapters 6, 9, or 10, it represents a nonzero, finite floating-point number.

Types of Notes

There are several types of notes used in *Inside Macintosh.*

Note

A note like this contains information that is interesting but possibly not essential to an understanding of the main text. u

IMPORTANT

A note like this contains information that is essential for an understanding of the main text. s

s **WARNING**

Warnings like this indicate potential problems that you should be aware of as you design your application. Failure to heed these warnings could result in system crashes or loss of data. s

For More Information

APDA is Apple’s worldwide source for over three hundred development tools, technical resources, training products, and information for anyone interested in developing applications on Apple platforms. Customers receive the quarterly *APDA Tools Catalog* featuring all current versions of Apple development tools and the most popular third-party development tools. Ordering is easy; there are no membership fees, and application forms are not required for most of our products. APDA offers convenient payment and shipping options, including site licensing.

To order products or to request a complimentary copy of the *APDA Tools Catalog*, contact

APDA
Apple Computer, Inc.
P.O. Box 319
Buffalo, NY 14207-0319

Telephone	800-282-2732 (United States) 800-637-0029 (Canada) 716-871-6555 (International)
Fax	716-871-6511
AppleLink	APDA
America Online	APDA
CompuServe	76666,2405
Internet	APDA@applelink.apple.com

If you provide commercial products and services, call 408-974-4897 for information on the developer support programs available from Apple.

For information of registering signatures, file types, Apple events, and other technical information, contact

Macintosh Developer Technical Support
Apple Computer, Inc.
20525 Mariani Avenue, M / S 75-3T
Cupertino, CA 95014-6299

PART ONE

The PowerPC Numerics Environment

This part is a general description of PowerPC Numerics. Chapter 1 describes the standards for floating-point arithmetic that PowerPC Numerics implements (IEEE Standard 754 and the FPCE technical report) and discusses why these standards are important. If you are unfamiliar with how computers perform floating-point arithmetic, you should read Chapter 1. Chapters 2 through 6 describe how PowerPC Numerics implements the standards. They describe the basic features shared by all PowerPC Numerics implementations, including

- the numeric data formats
- the special values NaN (Not-a-Number) and Infinity
- the methods by which floating-point expressions are evaluated
- environmental controls, such as setting the rounding direction and handling exceptions
- conversions between the different numeric formats
- operations supported by PowerPC Numerics

Although Part 1 uses the C programming language in its examples, many of the facilities of PowerPC Numerics are accessible to users of virtually any high-level programming language, as well as to assembly-language programmers.

CHAPTER 1

IEEE Standard Arithmetic

Contents

This chapter describes why IEEE standard floating-point arithmetic is important and why you should use it when programming. PowerPC Numerics is an implementation of the IEEE Standard 754 for binary floating-point arithmetic as well as the standard proposed by the Floating-Point C Extensions (FPCE) branch of the Numerical C Extensions Group (NCEG). This chapter explains the benefits that PowerPC Numerics provides by conforming to these standards. It provides an overview of both the IEEE and the FPCE recommendations—describing the scope of these standards and explaining how following them improves the accuracy of your programs. It provides some examples to demonstrate how much easier programming is when the standards are followed. Finally, it describes in general how PowerPC Numerics differs from the Standard Apple Numerics Environment (SANE).

You should read this chapter if you are unfamiliar with IEEE Standard 754 or the FPCE technical report and you want to find out more about them. If you are already familiar with these standards but you would like to find out how PowerPC Numerics implements them, you can skip to the next chapter.

About the IEEE Standard

PowerPC Numerics is a floating-point environment that complies with IEEE Standard 754. There are two IEEE standards for floating-point arithmetic: **IEEE Standard 754** for binary floating-point arithmetic and **IEEE Standard 854** for radix-independent floating-point arithmetic. When you see the term **IEEE standard** in this book without a number following, it means IEEE Standard 754.

The IEEE standards ensure that computers represent real numbers as accurately as possible and that computers perform arithmetic on real numbers as accurately as possible. Although there are infinitely many real numbers, a computer can represent only a finite number of them. Computers represent real numbers as **binary floating-point numbers**. Binary floating-point numbers can represent real numbers exactly in only a relatively few cases; in all other cases the representation is approximate. For example, 1/2 (0.5 in decimal) can be represented exactly in binary as 0.1. Other real numbers that can be represented exactly in decimal have repeating digits in binary and hence cannot be represented exactly, as shown in Table 1-1. For example, 1/10, or decimal 0.1 exactly, is 0.000110011 . . .in binary. Errors of this kind are unavoidable in any computer approximation of real numbers. Because of these errors, sums of fractions are often slightly incorrect. For example, 4/3 – 5/6 is not exactly equal to 1/2 on any computer, even on computers that use IEEE standard arithmetic.

Table 1-1 Approximation of real numbers

Fraction	Decimal approximation*	Binary approximation†
1 / 10	0.1000000000‡	0.000110011001100110011001101
1 / 2	0.5000000000‡	0.100000000000000000000000‡
4 / 3	1.333333333	1.01010101010101010101011
5 / 6	0.8333333333	0.110101010101010101010101
4 / 3 – 5 / 6	0.4999999997	0.100000000000000000000001

* 10 significant digits
† 23 significant digits
‡ Exact value

The IEEE standard defines data formats for floating-point numbers, shows how to interpret these formats, and specifies how to perform operations (known as **floating-point operations**) on numbers in these formats. It requires the following types of floating-point operations:

- basic arithmetic operations (add, subtract, multiply, divide, square root, remainder, and round-to-integer)
- conversion operations, which convert numbers to and from the floating-point data formats
- comparison operations, such as less than, greater than, and equal to
- environmental control operations, which manipulate the floating-point environment

The IEEE standard requires that the basic arithmetic operations have the following attributes:

- The result must be accurate in the precision in which the operation is performed. When a numerics environment is performing a floating-point operation, it calculates the result to a predetermined number of binary digits. This number of digits is called the **precision**. The result must be correct to the last binary digit.
- If the result cannot be represented exactly in the destination data format, it must be changed to the closest value that can be represented, using **rounding**. See the section “Careful Rounding” on page 1-5 for more information on why careful rounding is important.
- If an invalid input is provided or if the result cannot be represented exactly, a floating-point **exception** must be raised. See the section “Exception Handling” on page 1-6 for a description of why exception handling is important in floating-point arithmetic.

Starting to Use IEEE Arithmetic

You can get the benefit of much of the IEEE standard without special programming techniques; you simply use the floating-point variable formats and operations available in the programming language in which you are working, and the computer takes care of the rest. Other features might require changes to your applications. If you are new to numerical programming, you should approach the IEEE standard features in three stages:

1. Recompile your old programs with no changes; you will get many of the benefits.
2. Make small changes to obtain more benefits. For example, at this stage you might remove all code that tests for division by zero.
3. Use the advanced features, such as environmental controls, for special applications.

If you already use the IEEE standard features but your application is written for a non-Macintosh computer, see Appendix B, "Porting Programs to PowerPC Numerics."

Careful Rounding

If the result of an IEEE arithmetic operation cannot be represented exactly in binary format, the number is rounded. IEEE arithmetic normally rounds results to the nearest value that can be represented in the chosen data format. The difference between the exact result and the represented result is the **roundoff error**.

The IEEE standard requires that users be able to choose to round in directions other than to the nearest value. For example, sometimes you might want to know that rounding has not invalidated a computation. One way to do that would be to force the rounding direction so that you can be sure your results are higher (or lower) than the exact answer. Because it conforms to the IEEE standard, PowerPC Numerics gives you a means of doing that. Fully developed, this strategy is called *interval arithmetic* (Kahan 1980). For complete details on rounding directions, see Chapter 4, "Environmental Controls."

The following example is a simple demonstration of the advantages of careful rounding. Suppose your application performs operations that are mutually inverse; that is, operations $y = f(x)$, $x = g(y)$, such that $g(f(x)) = x$. There are many such operations, such as

$$y = x^2, \quad x = \sqrt{y}$$

$$y = 375x, \quad x = y/375$$

Suppose $F(x)$ is the computed value of $f(x)$, and $G(y)$ is the computed value of $g(y)$. Because many numbers cannot be represented exactly in binary, the computed values $F(x)$ and $G(y)$ will often differ from $f(x)$ and $g(y)$. Even so, if both functions are continuous and well behaved, and if you always round $F(x)$ and $G(y)$ to the nearest value, you might expect your computer arithmetic to return x when it performs the cycle of inverse operations, $G(F(x))$. It is difficult to predict when this relation will hold for computer numbers. Experience with other computers says it is too much to expect, but IEEE arithmetic very often returns the correct inverse value.

The reason for IEEE arithmetic's good behavior with respect to inverse operations is that it rounds so carefully. Even with all operations in, say, single precision, it evaluates the expression 3 × 1 / 3 to 1.0 exactly; some computers that do not follow the standard do not evaluate this expression exactly. If you find that surprising, you might enjoy running the code example in Listing 1-1 on a computer that does not use IEEE arithmetic and then on a PowerPC processor-based Macintosh computer. The default rounding provided by the numerics environment gives good results; the PowerPC processor-based Macintosh computer prints "No failures." The program will fail on a computer that doesn't have IEEE arithmetic—in particular, that doesn't round halfway cases in the same way that the IEEE standard's default rounding direction mode does.

Listing 1-1 Inverse operations

```
#include <stdio.h>
main()
{
   float x, y, a, b;
   int ix, iy,
   int nofail = 1;      /* Boolean, initialized to true */

   for (ix = 1; ix <= 12; ix++) {
      if ((ix != 7) && (ix != 11)) {      /* x is a sum of powers of two */
         for (iy = 1; iy <= 50; iy++) {
            x = ix;
            y = iy;
            a = y / x;
            b = x * a;        /* b == (x * y / x) == y */
            if (b != y) {
               nofail = 0;    /* false */
               printf("It failed for x = %d, y = %d\n", ix, iy);
            }
         }
      }
   }
   if (nofail) printf("No failures\n");
}
```

Exception Handling

The IEEE standard defines five exceptions that indicate when an exceptional event has occurred. They are

- invalid operation
- underflow

- overflow
- division by zero
- inexact result

There are three ways your application can deal with exceptions:

- Continue operation.
- Stop on exceptions, if you think they will invalidate your results.
- Include code to do something special when exceptions happen.

The IEEE standard lets programs deal with the exceptions in reasonable ways. It defines the special values NaN and Infinity, which allow a program to continue operation; see the section “Interpreting Floating-Point Values” in Chapter 2, “Floating-Point Data Formats.” The IEEE standard also defines exception flags, which a program can test to detect exceptional events.

IEEE arithmetic allows the option to stop computation when exceptional events arise, but there are good reasons why you might prefer not to have to stop. The following examples illustrate some of those reasons.

Example: Finding Zero Return Values

Suppose you want to find the first positive integer that causes a function to cross the x-axis. A simple version of the code might look like this:

```
for (i = 0; i < MAXVALUE; i++)
   if (func(i) == 0)
      printf("It crosses when x = %g\n", i);
```

Further, suppose that `func` was defined like this:

```
double func(double x)
{
   return(sqrt(x - 3));
}
```

The intent of the `for` loop is to find out where the function crosses the x-axis and print out that information; it does not really care about the value returned from `func` unless the value is 0. However, this loop will fail when `i` is less than 3 because you cannot take the square root of a negative number. With a C compiler that supports PowerPC Numerics, performing the square root operation on a negative number returns a NaN, allowing the loop to produce the desired result. To obtain the desired result on all computers, something more cumbersome would have to be written. By allowing the square root of a negative number, PowerPC Numerics allows more straightforward code.

This program fragment demonstrates the principal service performed by NaNs: they permit deferred judgments about variables whose values might be unavailable (that is, uninitialized) or the result of invalid operations. Instead of having the computer stop a computation as soon as a NaN appears, you might prefer to have it continue, if whatever caused the NaN is irrelevant to the solution.

Example: Searching Without Stopping

Suppose a program has to search through a database for a maximum value that has to be calculated. The search loop might call a subroutine to perform some calculation on the data in each record and return a value for the program to test or compare. The code might look like this:

```
max = –INFINITY;
for (i = 0; i < MAXRECORDS; i++)
   if((temp = computation(record[i].value)) > max)
      max = temp;
```

Suppose that the `value` field of the `record` structure is not a required field when the data is entered, so that for some records, data might be nonexistent or invalid. In many machines, that would cause the program to stop. To avoid having the program stop during the search, you would have to add tests for all the exceptional cases. With PowerPC Numerics, the subroutine `computation` does not stop for nonexistent or invalid data; it simply returns a NaN.

This is another example of the way arithmetic that includes NaNs allows the program to ignore irrelevancies, even when they cause invalid operations. Using arithmetic without NaNs, you would have to anticipate all exceptional cases and add code to the program to handle every one of them in advance. With NaNs, you can handle all exceptional cases after they have occurred, or you can simply ignore them, as in this example.

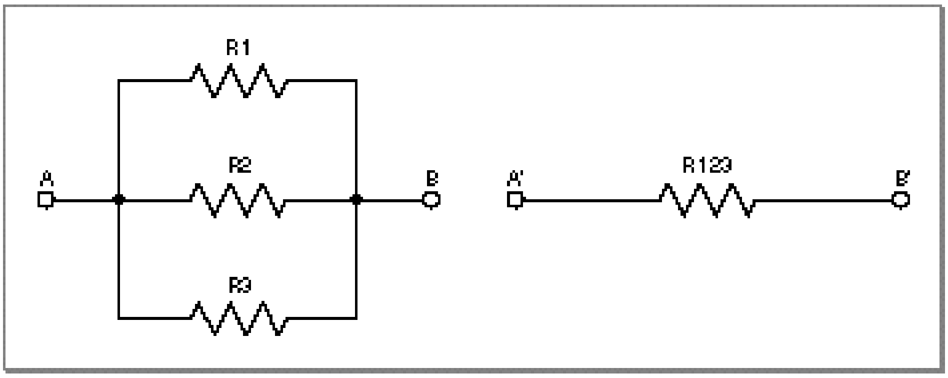
Example: Parallel Resistances

Like NaNs, Infinities enable the program to handle cases that otherwise would require special programming to keep from stopping. Here is an example where arithmetic with Infinities is entirely reasonable.

When three electrical resistances R1, R2, and R3 are connected in parallel, as shown in Figure 1-1, their effective resistance is the same as a single resistance whose value R123 is given by this formula:

$$R123 = \frac{1}{\frac{1}{R1} + \frac{1}{R2} + \frac{1}{R3}}$$

Figure 1-1 Parallel resistances



The formula gives correct results for positive resistance values between 0 (corresponding to a short circuit) and ∞ (corresponding to an open circuit) inclusive. On computers that do not allow division by zero, you would have to add tests designed to filter out the cases with resistance values of zero. (Negative values can cause trouble for this formula, regardless of the style of the arithmetic, but that reflects their troublesome nature in circuits, where they can cause instability.)

Arithmetic with Infinities usually gives reasonable results for expressions in which each independent variable appears only once.

Using IEEE Arithmetic

This section provides some example computations and describes how using IEEE arithmetic in the PowerPC Numerics environment makes programming these computations easier.

Evaluating Continued Fractions

Consider a typical continued fraction $\mathrm{cf}(x)$.

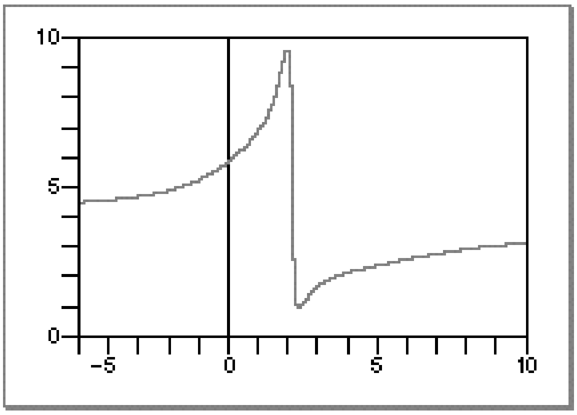
$$\mathrm{cf}(x) = 4 - \cfrac{3}{x - 2 - \cfrac{1}{x - 7 + \cfrac{10}{x - 2 - \cfrac{2}{x - 3}}}}$$

An algebraically equivalent expression is $\mathrm{rf}(x)$:

$$\mathrm{rf}(x) = \frac{622 - x(751 - x(324 - x(59 - 4x)))}{112 - x(151 - x(72 - x(14 - x)))}$$

Both expressions represent the same rational function, one whose graph is smooth and unexceptional, as shown in Figure 1-2.

Figure 1-2 Graph of continued fraction functions $\mathrm{cf}(x)$ and $\mathrm{rf}(x)$



Although the two functions $\mathrm{rf}(x)$ and $\mathrm{cf}(x)$ are equal, they are not computationally equivalent. For instance, consider $\mathrm{rf}(x)$ at the following values of x:

$x = 1$	$\mathrm{rf}(1) = 7$
$x = 2$	$\mathrm{rf}(2) = 4$
$x = 3$	$\mathrm{rf}(3) = 8/5$
$x = 4$	$\mathrm{rf}(4) = 5/2$

Whereas $\mathrm{rf}(x)$ is perfectly well behaved, those values of x lead to division by zero when computing $\mathrm{cf}(x)$ and cause many computers to stop. In IEEE standard arithmetic, division by zero produces an Infinity. Therefore, PowerPC Numerics has no difficulty in computing $\mathrm{cf}(x)$ for those values.

On the other hand, simply computing $\mathrm{rf}(x)$ instead of $\mathrm{cf}(x)$ can also cause problems. If the absolute value of x is so big that x^4 overflows the chosen data format, then $\mathrm{cf}(x)$ approaches $\mathrm{cf}(\infty) = 4$ but computing $\mathrm{rf}(x)$ encounters $(\mathrm{overflow})/(\mathrm{underflow})$, which yields something else. PowerPC Numerics returns NaN for such cases; some other machines get $(\text{maximum value})/(\text{maximum value}) = 1$. Also, at arguments x between 1.6 and 2.4, the formula $\mathrm{rf}(x)$ suffers from roundoff error much more than $\mathrm{cf}(x)$ does. For those reasons, computing $\mathrm{cf}(x)$ is preferable to computing $\mathrm{rf}(x)$ if division by zero works the way it does in PowerPC Numerics, that is, if it produces Infinity instead of stopping computation.

In general, division by zero is an exceptional event not merely because it is rare but because different applications require different consequences. If you are not satisfied with the consequences supplied by the default PowerPC Numerics environment, you can choose other consequences by making the program test for NaNs and Infinities (or for the flags that signal their creation).

Rather than sprinkle tests throughout the program in an attempt to keep exceptions from occurring, you might prefer to put one or two tests near the end of the code to detect the (rare) occurrence of an exception and modify the results appropriately. That is more economical than testing every divisor for zero (since zero divisors are rare).

Computing the Area of a Triangle

Here is a familiar and straightforward task that fails when subtraction is aberrant: Compute the area $A(x, y, z)$ of a triangle given the lengths x, y, z of its sides. The formula given here performs this calculation almost as accurately as its individual floating-point operations are performed by the computer it runs on, provided the computer does not drop digits prematurely during subtraction. The formula works correctly, and provably so, on a wide range of machines, including all implementations of PowerPC Numerics.

The classical formula, attributed to Heron of Alexandria, is

$$A(x, y, z) = \sqrt{s(s-x)(s-y)(s-z)}$$

where $s = (x + y + z)/2$.

For needle-shaped triangles, that formula gives incorrect results on computers *even when every arithmetic operation is correctly rounded.* For example, Table 1-2 shows an extreme case with results rounded to five decimal digits. With the values shown, rounded $(x + (y + z))/2$ must give either 100.01 or 100.02. Substituting those values for s in Heron's formula yields either 0.0 or 1.5813 instead of the correct value 1.000025.

Evidently, Heron's formula would be a very bad way to calculate ratios of areas of nearly congruent needle-shaped triangles.

Table 1-2 Area using Heron's formula

	Correct	**Rounding downward**	**Rounding upward**
x	100.01	100.01	100.01
y	99.995	99.995	99.995
z	0.025	0.025	0.025
$(x+(y+z))/2$	100.015	100.01	100.02
A	1.000025	0.0000	1.5813

A good procedure, numerically stable on machines that do not truncate prematurely during subtraction (such as machines that use IEEE arithmetic), is the following:

1. Sort x, y, z so that $x \geq y \geq z$.
2. Test for $z \geq x - y$ to see whether the triangle exists.
3. Compute A by the formula

$$A = \sqrt{((x + (y + z))(z - (x - y))(x + (y - z)))/4}$$

s **WARNING**
This formula works correctly only if you do not remove any of the parentheses. s

The success of the formula depends upon the following easily proved theorem:

THEOREM *If p and q are represented exactly in the same conventional floating-point format, and if* $1/2 \leq p/q \leq 2$, *then* $p - q$ *too is representable exactly in the same format (unless* $p - q$ *suffers underflow, something that cannot happen in IEEE arithmetic).*

The theorem merely confirms that subtraction is exact when massive cancellation occurs. That is why each factor inside the square root expression is computed correctly to within a unit or two in its last digit kept, and *A* is not much worse, on computers that subtract the way PowerPC Numerics does. On machines that flush tiny results to zero, this formula for *A* fails because $(p - q)$ can underflow.

About the FPCE Technical Report

Even though many computers now conform to the IEEE standard, the standard has suffered from a lack of high-level portability. The reason is that the standard does not define bindings to high-level languages; it only defines a programming environment. For instance, the standard defines data formats that should be supported but does not tell how these data formats should map to variable types in high-level languages. It also specifies that the user must be able to control rounding direction but falls short of defining how the user is able to do so.

However, the definition of a binding is in progress for the C programming language. The Floating-Point C Extensions (FPCE) branch of the Numerical C Extensions Group (NCEG), or **ANSI X3J11.1**, has proposed a general floating-point specification for the C programming language, called the **FPCE technical report**, that contains additional specifications for implementations that comply with IEEE floating-point standards 754 and 854.

The FPCE technical report not only specifies how to implement the requirements of the IEEE standards, but also requires some additional functions, called **transcendental functions** (sometimes called elementary functions). These functions are consistent with the IEEE standard and can be used as building blocks in numerical functions. The transcendental functions include the usual logarithmic and exponential functions, as well as $\ln(1 + x)$ and $e^x - 1$; financial functions for compound interest and annuity calculations; trigonometric functions; error and gamma functions; and a random number generator. The **PowerPC Numerics library**, contained in the file MathLib, implements the transcendental functions.

Part 2 of this book describes how PowerPC Numerics complies with the recommendations in the FPCE technical report.

PowerPC Numerics Versus SANE

Although PowerPC Numerics is an implementation of the IEEE Standard, it is not the **Standard Apple Numerics Environment (SANE)**. SANE is the numerics environment used on **680x0-based Macintosh computers**, and it is the numerics environment used when you run a 680x0 application on a PowerPC processor-based Macintosh computer. PowerPC Numerics is the environment used when you run an application built for a PowerPC processor-based Macintosh computer.

There are fundamental differences between PowerPC Numerics and SANE because of the differences in the microprocessors on which the two environments are used. The major difference is that SANE supports an 80-bit extended type and performs all floating-point computations in extended precision. This protects the user from roundoff error, overflows, and underflows that might occur in an intermediate value when determining the result of an expression. Because the PowerPC processor is double-based, support of an 80-bit data type would be inefficient. It instead supports a 128-bit type (in software) called double-double (which corresponds to the `long double` type in C). PowerPC Numerics provides this wide type only for cases where precision greater than that provided by the double format is necessary; PowerPC Numerics does not perform all computations in double-double precision. Instead, PowerPC Numerics recommends a method by which an expression is evaluated in the widest precision necessary (see Chapter 3, “Expression Evaluation”).

Another fundamental difference is that PowerPC Numerics conforms to the FPCE recommendations as well as to the IEEE standard. C implementations using SANE do not necessarily comply with the FPCE recommendations.

See Appendix A, “SANE Versus PowerPC Numerics,” for more information on the differences between PowerPC Numerics and SANE.

Floating-Point Data Formats

Contents

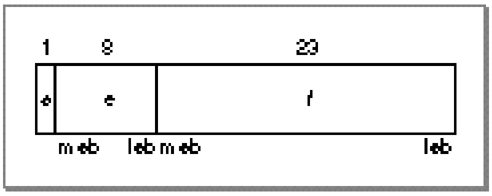
This chapter describes the data formats your PowerPC application can use to represent floating-point numbers. It begins by discussing in general the methods PowerPC Numerics uses to store and interpret floating-point values and by explaining why those methods were chosen. The chapter introduces the special values zero, NaN (Not-a-Number), and Infinity and explains why these special values are necessary. Next is an in-depth description of the numeric data formats with a discussion of how these formats represent floating-point values. At the end of the chapter, you will find a table comparing the size, range, and precision of the numeric data formats. This table can help you choose which data format is best for your application.

You should read this chapter to learn about the floating-point data formats available on PowerPC processor-based Macintosh computers and to learn more about how your computer encodes and manipulates floating-point numbers.

About Floating-Point Data Formats

The IEEE standard defines several floating-point data formats, some required and others recommended. IEEE requires that each data format have a **sign bit** (*s*), an **exponent** field (*e*), and a **fraction** field (*f*). For each format, it lists requirements for the minimum lengths of these fields. For example, the standard describes a 32-bit single format whose exponent field must be 8 bits long and whose fraction field must be 23 bits long. Figure 2-1 shows the IEEE requirements for the single format. (In this figure, *msb* stands for most significant bit and *lsb* stands for least significant bit.)

Figure 2-1 IEEE single format



The only required data format is the 32-bit single format. A 64-bit double format is strongly recommended. The IEEE standard also describes two data formats called single-extended and double-extended and recommends that floating-point environments provide the extended format corresponding to the widest basic format (single or double) they support.

To conform to the IEEE requirements on floating-point data formats, the PowerPC Numerics environment provides three data formats: single (32 bits), double (64 bits), and double-double (128 bits). The single and double formats are implemented exactly as described in the standard. The double-double format is provided in place of the recommended double-extended format. IEEE requires that the double-extended format be at least 79 bits long with at least a 15-bit exponent. The double-double format is

128 bits long and has an 11-bit exponent. The double-double format is just what its name sounds like: two double-format numbers combined. The PowerPC assembly-language multiply-add instructions, which multiply two double-format numbers and add a third with at most one roundoff error, make implementing the double-double format much more efficient than implementing a true IEEE double-extended format. See Chapter 14, “Assembly-Language Numeric Operations,” for more information on the multiply-add instructions.

Table 2-1 shows how the three numeric data formats correspond to C variable types. For more information about data types in C, refer to Chapter 7, “Numeric Data Types in C.”

Table 2-1 Names of data types

PowerPC Numerics data format	C type
IEEE single	`float`
IEEE double	`double`
Double-double	`long double`

The IEEE standard also makes requirements about how the values in these data formats are interpreted. PowerPC Numerics follows these requirements exactly. They are described in the next section.

Interpreting Floating-Point Values

Regardless of which data format (single, double, or double-double) you use, the numerics environment uses the same basic method to interpret which floating-point value it represents. This section describes that method.

Every floating-point data format has a sign bit, an exponent field, and a fraction field. These three fields provide binary encodings of a sign (+ or –), an exponent, and a **significand**, respectively, of a floating-point value. The value is interpreted as

$$\pm\ significand \times 2^{exponent\ -\ bias}$$

where

± is the sign stored in the sign bit (1 is negative, 0 is positive).

significand has the form $b_0 . b_1 b_2 b_3 \ldots b_{precision\ -\ 1}$ where $b_1 b_2 b_3 \ldots b_{precision\ -\ 1}$ are the bits in the fraction field and b_0 is an implicit bit whose value is interpreted as described in the sections “Normalized Numbers” and “Denormalized Numbers.” The significand is sometimes called the *mantissa.*

exponent is the value of the exponent field.

bias is the bias of the exponent. The **bias** is a predefined value (127 for single format, 1023 for double and double-double formats) that is added to the exponent when it is stored in the exponent field. When the floating-point number is evaluated, the bias is subtracted to return the correct exponent. The minimum biased exponent field (all 0's) and maximum biased exponent field (all 1's) are assigned special floating-point values (described in the next several sections).

In a numeric data format, each valid representation belongs to exactly one of these classes, which are described in the sections that follow:

- normalized numbers
- denormalized numbers
- Infinities
- NaNs (signaling or quiet)
- zeros

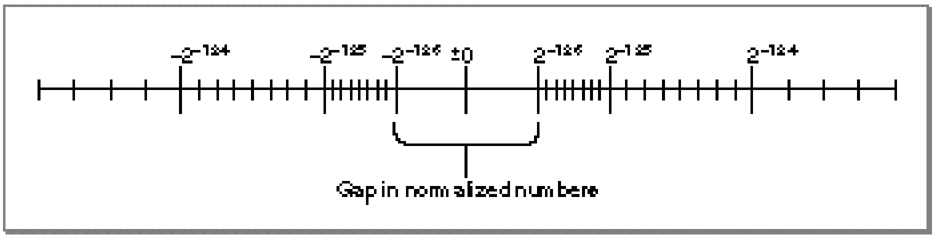
Normalized Numbers

The numeric data formats represent most floating-point numbers as **normalized numbers**, meaning that the implicit leading bit (b_0 on page 2-4) of the significand is 1. Normalization maximizes the resolution of the data type and ensures that representations are unique. Figure 2-2 shows the magnitudes of normalized numbers in single precision on the number line. The spacing of the vertical marks indicates the relative density of numbers in each binade. (A **binade** is a collection of numbers between two successive powers of 2.) Notice that the numbers get more dense as they approach 0.

Note

The figure shows only the relative density of the numbers; in reality, the density is immensely greater than it is possible to show in such a figure. For example, there are 2^{23} (8,388,608) single-precision numbers in the interval $2^{-126} \le x < 2^{-125}$. u

Figure 2-2 Normalized single-precision numbers on the number line



Using only normalized representations creates a gap around the value 0, as shown in Figure 2-2. If a computer supports only the normalized numbers, it must round all tiny values to 0. For example, suppose such a computer must perform the operation $x - y$, where x and y are very close to, but not equal to, each other. If the difference between x and y is smaller than the smallest normalized number, the computer must deliver 0 as the result. Thus, for such **flush-to-zero systems**, the following statement is *not* true for all real numbers:

$x - y = 0$ if and only if $x = y$

Denormalized Numbers

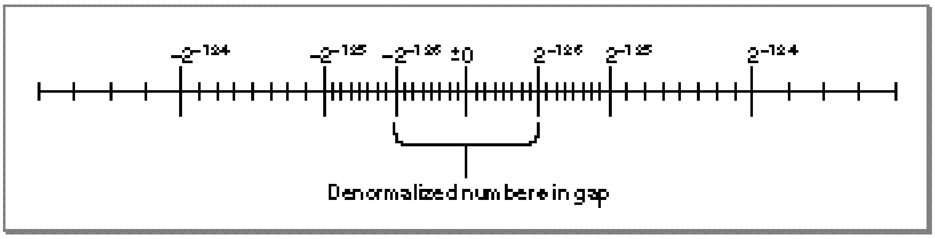
Instead of using only normalized numbers and allowing this small gap around 0, PowerPC processor-based Macintosh computers use **denormalized numbers**, in which the leading implicit bit (b_0 on page 2-4) of the significand is 0 and the minimum exponent is used.

Note

Some references use the term **subnormal numbers** instead of denormalized numbers. u

Figure 2-3 illustrates the relative magnitudes of normalized and denormalized numbers in single precision. Notice that the denormalized numbers have the same density as the numbers in the smallest normalized binade. This means that the roundoff error is the same regardless of whether an operation produces a denormalized number or a very small normalized number. As stated previously, without denormalized numbers, operations would have to round tiny values to 0, which is a much greater roundoff error.

Figure 2-3 Denormalized single-precision numbers on the number line



To put it another way, the use of denormalized numbers makes the following statement true for all real numbers:

$x - y = 0$ if and only if $x = y$

Another advantage of denormalized numbers is that error analysis involving small values is much easier without the gap around zero shown in Figure 2-2 (Demmel 1984).

The computer determines that a floating-point number is denormalized (and therefore if its implicit leading bit is interpreted as 0) when the biased exponent field is filled with 0's and the fraction field is nonzero.

Table 2-2 shows how a single-precision value A_0 becomes progressively denormalized as it is repeatedly divided by 2, with rounding to nearest. This process is called **gradual underflow**. In the table, values $A_2 \ldots A_{25}$ are denormalized; A_{25} is the smallest positive denormalized number in single format. Notice that as soon as the values are too small to be normalized, the biased exponent value becomes 0.

Table 2-2 Example of gradual underflow

Variable or operation	Value	Biased exponent	Comment
A_0	1.100 1100 1100 1100 1100 1101 × 2^{-125}	2	
$A_1 = A_0/2$	1.100 1100 1100 1100 1100 1101 × 2^{-126}	1	
$A_2 = A_1/2$	0.110 0110 0110 0110 0110 0110 × 2^{-126}	0	Inexact*
$A_3 = A_2/2$	0.011 0011 0011 0011 0011 0011 × 2^{-126}	0	Exact result
$A_4 = A_3/2$	0.001 1001 1001 1001 1001 1010 × 2^{-126}	0	Inexact*
	.		
	.		
	.		
$A_{23} = A_{22}/2$	0.000 0000 0000 0000 0000 0011 × 2^{-126}	0	Exact result
$A_{24} = A_{23}/2$	0.000 0000 0000 0000 0000 0010 × 2^{-126}	0	Inexact*
$A_{25} = A_{24}/2$	0.000 0000 0000 0000 0000 0001 × 2^{-126}	0	Exact result
$A_{26} = A_{25}/2$	0.0	0	Inexact*

* Whenever division returns an inexact tiny value, the exception bit for underflow is set to indicate that a low-order bit has been lost.

Infinities

An **Infinity** is a special bit pattern that can arise in one of two ways:

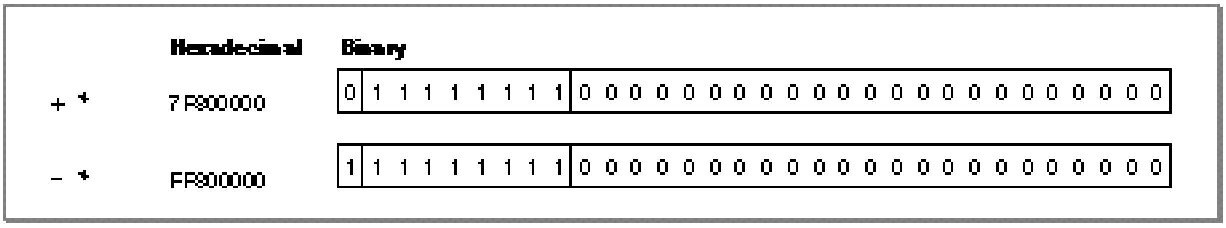
- When an operation (such as 1/0) should produce a mathematical infinity, the result is an Infinity.
- When an operation attempts to produce a number with a magnitude too great for the number's intended floating-point data type, the result might be a value with the largest possible magnitude or it might be an Infinity (depending on the current rounding direction).

These bit patterns (as well as NaNs, introduced next) are recognized in subsequent operations and produce predictable results. The Infinities, one positive and one negative, generally behave as suggested by the theory of limits. For example:

- Adding 1 to +∞ yields +∞.
- Dividing −1 by +0 yields −∞.
- Dividing 1 by −∞ yields −0.

The computer determines that a floating-point number is an Infinity if its exponent field is filled with 1's and its fraction field is filled with 0's. So, for example, in single format, if the sign bit is 1, the exponent field is 255 (which is the maximum biased exponent for the single format), and the fraction field is 0, the floating-point number represented is −∞ (see Figure 2-4).

Figure 2-4 Infinities represented in single precision



	Hexadecimal	Binary		
+ ∞	7F800000	0	1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
− ∞	FF800000	1	1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

NaNs

When a numeric operation cannot produce a meaningful result, the operation delivers a special bit pattern called a **NaN (Not-a-Number)**. For example, zero divided by zero, +∞ added to −∞, and $\sqrt{-1}$ yield NaNs. A NaN can occur in any of the numeric data formats (single, double, and double-double), but generally, system-specific **integer types** (non-numeric types exclusively for integer values) have no representation for NaNs.

NaNs propagate through arithmetic operations. Thus, the result of 3.0 added to a NaN is the same NaN. If two operands of an operation are NaNs, the result is one of the NaNs. NaNs are of two kinds: **quiet NaNs**, the usual kind produced by floating-point operations, and **signaling NaNs**.

When a signaling NaN is encountered as an operand of an arithmetic operation, the invalid-operation exception is signaled and a quiet NaN is the delivered result. Signaling NaNs are not created by any numeric operations, but you might find it useful to create signaling NaNs manually. For example, you might fill uninitialized memory with signaling NaNs so that if one is ever encountered in a program, you will know that uninitialized memory is accessed.

A NaN may have an associated code that indicates its origin. These codes are listed in Table 2-3. The NaN code is the 8th through 15th most significant bits of the fraction field.

Table 2-3 NaN codes

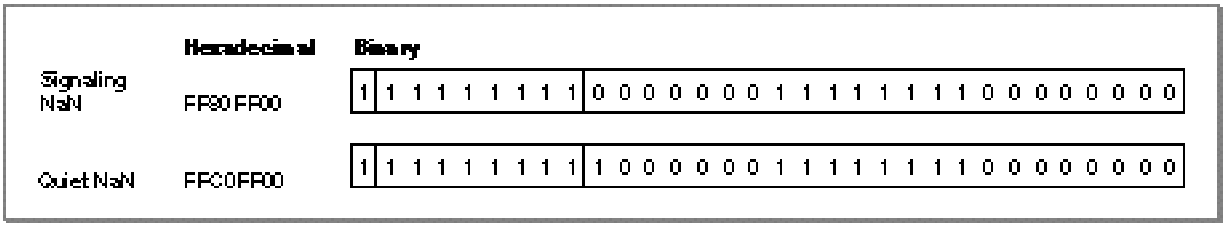
Decimal	Hexadecimal	Meaning
1	0x01	Invalid square root, such as $\sqrt{-1}$
2	0x02	Invalid addition, such as $(+\infty) + (-\infty)$
4	0x04	Invalid division, such as $0/0$
8	0x08	Invalid multiplication, such as $0 \times \infty$
9	0x09	Invalid remainder or modulo such as *x* rem 0
17	0x11	Attempt to convert invalid ASCII string
21	0x15	Attempt to create a NaN with a zero code
33	0x21	Invalid argument to trigonometric function (such as cos, sin, tan)
34	0x22	Invalid argument to inverse trigonometric function (such as acos, asin, atan)
36	0x24	Invalid argument to logarithmic function (such as log, log10)
37	0x25	Invalid argument to exponential function (such as exp, expm1)
38	0x26	Invalid argument to financial function (compound or annuity)
40	0x28	Invalid argument to inverse hyperbolic function (such as acosh, asinh)
42	0x2A	Invalid argument to gamma function (gamma or lgamma)

Note

The PowerPC processor always returns 0 for the NaN code. u

The computer determines that a floating-point number is a NaN if its exponent field is filled with 1’s and its fraction field is nonzero. The most significant bit of the fraction field distinguishes quiet and signaling NaNs. It is set for quiet NaNs and clear for signaling NaNs. For example, in single format, if the sign field has the value 1, the exponent field has the value 255, and the fraction field has the value 65,280, then the number is a signaling NaN. If the sign is 1, the exponent is 255, and the fraction field has the value 4,259,584 (which means the fraction field has a leading 1 bit), the value is a quiet NaN. Figure 2-5 illustrates these examples.

Figure 2-5 NaNs represented in single precision



Zeros

Each floating-point format has two representations for zero: +0 and −0. Although the two zeros compare as equal $(+0) = -0$, their behaviors in IEEE arithmetic are slightly different.

Ordinarily, the sign of zero does not matter except (possibly) for a function discontinuous at zero. Though the two forms are numerically equal, a program can distinguish +0 from −0 by operations such as division by zero or by performing the numeric copysign function.

The sign of zero obeys the usual sign laws for multiplication and division. For example, $(+0) \times (-1) = -0$ and $1/(-0) = -\infty$. Because extreme negative underflows yield −0, expressions like $1/x^3$ produce the correct sign for ∞ when x is tiny and negative. Addition and subtraction produce −0 only in these cases:

- $(-0) - (+0)$ yields -0
- $(-0) + (-0)$ yields -0

When rounding downward, with x finite,

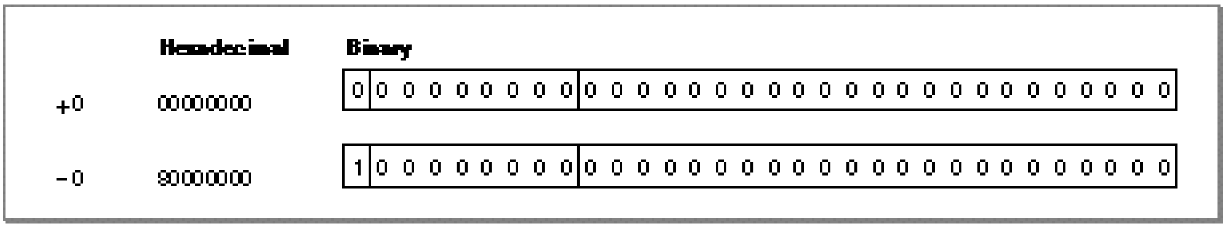
- $x - x$ yields -0
- $x + (-x)$ yields -0

The square root of −0 is −0.

The sign of zero is important in complex arithmetic (Kahan 1987).

The computer determines that a floating-point number is 0 if its exponent field and its fraction field are filled with 0’s. For example, in single format, if the sign bit is 0, the exponent field is 0, and the fraction field is 0, the number is +0 (see Figure 2-6).

Figure 2-6 Zeros represented in single precision



Formats

This section shows the three numeric data formats: single, double, and double-double. These are pictorial representations and might not reflect the actual byte order in any particular implementation.

Each of the diagrams on the following pages is followed by a table that gives the rules for evaluating the number. In each field of each diagram, the leftmost bit is the most significant bit (msb) and the rightmost is the least significant bit (lsb). Table 2-4 defines the symbols used in the diagrams.

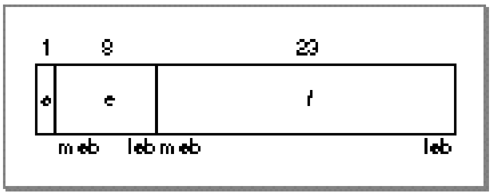
Table 2-4 Symbols used in format diagrams

Symbol	Description
v	Value of number
s	Sign bit
e	Biased exponent (*exponent* + *bias*)
f	Fraction (*significand* without leading bit)

Single Format

The 32-bit **single format** is divided into three fields having 1, 8, and 23 bits (see Figure 2-7).

Figure 2-7 Single format



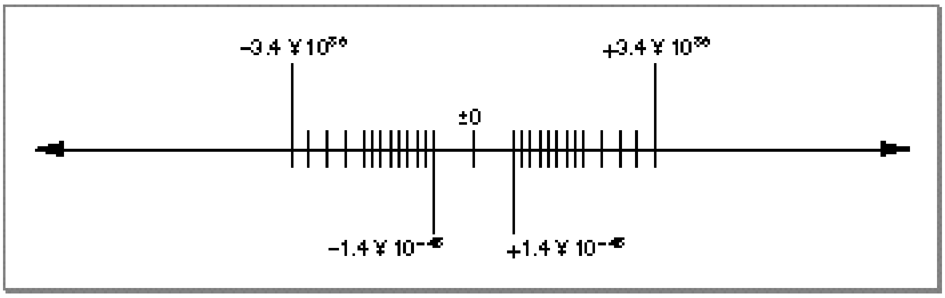
The interpretation of a single-format number depends on the values of the exponent field (*e*) and the fraction field (*f*), as shown in Table 2-5.

Table 2-5 Values of single-format numbers (32 bits)

If biased exponent *e* is:	And fraction *f* is:	Then value *v* is:	And the class of *v* is:
$0 < e < 255$	(any)	$v = (-1)^s \times 2^{(e-127)} \times (1.f)$	Normalized
$e = 0$	$f \neq 0$	$v = (-1)^s \times 2^{(-126)} \times (0.f)$	Denormalized
$e = 0$	$f = 0$	$v = (-1)^s \times 0$	Zero
$e = 255$	$f = 0$	$v = (-1)^s \times \infty$	Infinity
$e = 255$	$f \neq 0$	*v* is a NaN	NaN

Figure 2-8 shows the range and density of the real numbers that can be represented as single-format floating-point numbers using normalized and denormalized values. The vertical marks indicate the relative density of the numbers that can be represented. As explained in the section “Normalized Numbers” on page 2-5, the number of representable values gets more dense closer to 0.

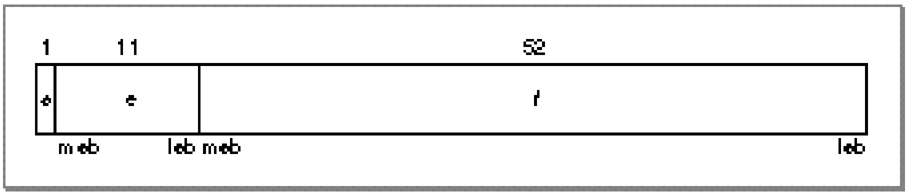
Figure 2-8 Single-format floating-point numbers on the real number line



Double Format

The 64-bit **double format** is divided into three fields having 1, 11, and 52 bits (see Figure 2-9).

Figure 2-9 Double format



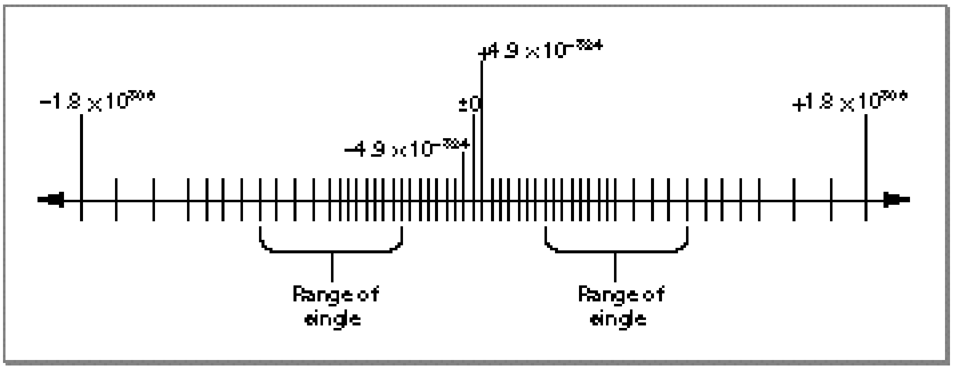
The interpretation of a double-format number depends on the values of the exponent field (*e*) and the fraction field (*f*), as shown in Table 2-6.

Table 2-6 Values of double-format numbers (64 bits)

If biased exponent *e* is:	And fraction *f* is:	Then value *v* is:	And the class of *v* is:
$0 < e < 2047$	(any)	$v = (-1)^s \times 2^{(e-1023)} \times (1.f)$	Normalized
$e = 0$	$f \neq 0$	$v = (-1)^s \times 2^{(-1022)} \times (0.f)$	Denormalized
$e = 0$	$f = 0$	$v = (-1)^s \times 0$	Zero
$e = 2047$	$f = 0$	$v = (-1)^s \times \infty$	Infinity
$e = 2047$	$f \neq 0$	*v* is a NaN	NaN

Figure 2-10 shows the range and density of the real numbers that can be represented as double-format floating-point numbers using normalized and denormalized values. The vertical marks indicate the relative density of the numbers that can be represented. As explained in the section “Normalized Numbers” on page 2-5, the number of representable values gets more dense closer to 0.

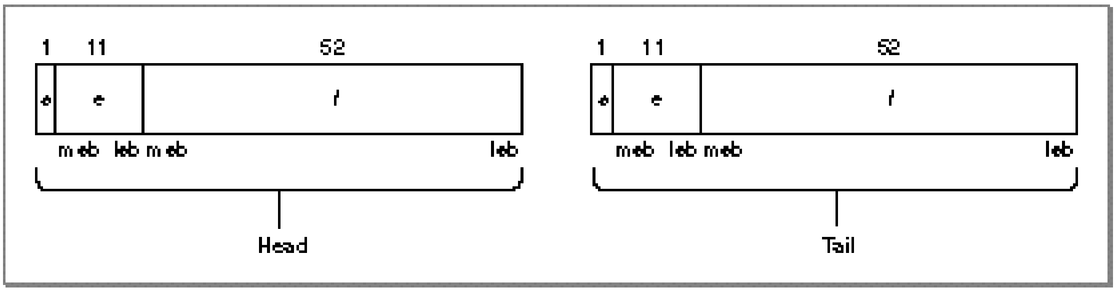
Figure 2-10 Double-format floating-point values on the real number line



Double-Double Format

The 128-bit **double-double format** is made up of two double-format numbers (see Figure 2-11).

Figure 2-11 Double-double format



The value of a double-double number is the sum of its head and tail components. These two components are both double numbers, and therefore the value of each component is determined as shown in Table 2-6. It is recommended that the tail's exponent be at least 54 less than the head's exponent. Numeric operations that produce double-double results always produce numbers in this form.

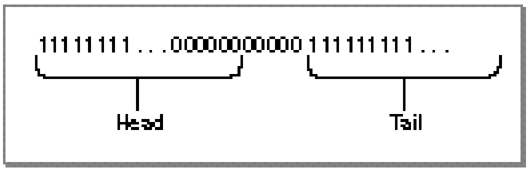
IMPORTANT

It is possible, but not recommended, to create a double-double format that does not follow this form. If you do not follow this form when creating a double-double number, the results are unpredictable. s

The requirement that the tail's exponent be at least 54 less than the head's exponent guarantees that the significand of the tail is more or less concatenated to the significand of the head (which is 53 bits long) when the two values are added together. For example, if the head component's exponent is 2^{200}, the tail component's exponent can be no greater than 2^{146}, so that in the value represented by this double-double format number, the head represents the first 53 binary digits and the tail represents the remaining digits.

Note that the difference between the exponent values may be greater than 54 and that the head and the tail can have different signs. To continue with the example, suppose the tail's exponent is 2^{140} instead of 2^{146}. The binary number represented would be as shown in Figure 2-12.

Figure 2-12 Double-double format number example



The head represents the binary places 2^{200} down to 2^{147}. The tail represents the binary places 2^{140} down to 2^{87}. The zeros between the head and the tail are necessary to represent the binary places 2^{146} to 2^{141}. This particular number has 112 units of precision—53 units from the head, 53 from the tail, and 6 units between the head and the tail. The double-double format always has at least 107 bits of precision, and if the tail's exponent is more than 54 less than the head's exponent, it has even greater precision.

If the value of the head component is a normalized number, then the value of the double-double number is the sum of the head and the tail. In the recommended form, if the head is not a normalized number (meaning it is denormalized, 0, NaN, or Infinity), the head contains the value of the double-double number, and the tail contains 0. This way, when you add the head and the tail, you still get the value of the head.

Although the precision of the double-double format is much greater than that of the double format, the range of the two formats is the same. However, because the double-double format is implemented in software, this format is much slower to use than the double format. Because of this, you should always use the double format unless you need the extra precision provided by the double-double format.

Range and Precision of Data Formats

Table 2-7 shows the precision, range, and memory usage for each numeric data format. You can use this table to compare the data formats and choose which one is needed for your application. Typically, choosing a data format requires that you determine the tradeoffs between

- fixed-point or floating-point form
- precision
- range
- memory usage
- speed

In the table, decimal ranges are expressed as rounded, two-digit decimal representations of the exact binary values. The speed of a given data format varies depending on the particular implementation of PowerPC Numerics. (See Chapter 5, “Conversions,” for information on aspects of conversion relating to precision.)

Table 2-7 Summary of PowerPC Numerics data formats

	Single	**Double**	**Double-double**
Size (bytes:bits)	4:32	8:64	16:128
Range of binary exponents			
Minimum	−126	−1022	−1022
Maximum	127	1023	1023
Significand precision			
Bits	24	53	≥ 107
Decimal digits	7–8	15–16	≥ 32
Decimal range approximate			
Maximum positive	$3.4 \times 10^{+38}$	$1.8 \times 10^{+308}$	$1.8 \times 10^{+308}$
Minimum positive norm	1.2×10^{-38}	2.2×10^{-308}	2.2×10^{-308}
Minimum positive denorm	1.4×10^{-45}	4.9×10^{-324}	4.9×10^{-324}
Maximum negative denorm	-1.4×10^{-45}	-4.9×10^{-324}	-4.9×10^{-324}
Maximum negative norm	-1.2×10^{-38}	-2.2×10^{-308}	-2.2×10^{-308}
Minimum negative	$-3.4 \times 10^{+38}$	$-1.8 \times 10^{+308}$	$-1.8 \times 10^{+308}$

For example, in single format, the largest representable number is composed as follows:

$$\begin{aligned}
\textit{significand} &= (2 - 2^{-23}) \\
&= 1.11111111111111111111111_2 \\
\text{exponent} &= 127 \\
\text{value} &= (2 - 2^{-23}) \times 2^{127} \\
&\approx 3.403 \times 10^{38}
\end{aligned}$$

The smallest positive normalized number representable in single format is made up as follows:

$$\begin{aligned}
\textit{significand} &= 1 \\
&= 1.00000000000000000000000_2 \\
\text{exponent} &= -126 \\
\text{value} &= 1 \times 2^{-126} \\
&\approx 1.175 \times 10^{-38}
\end{aligned}$$

For denormalized numbers, the smallest positive value representable in the single format is made up as follows:

$$\begin{aligned}
\textit{significand} &= 2^{-23} \\
&= 0.00000000000000000000001_2 \\
\text{exponent} &= -126 \\
\text{value} &= 2^{-23} \times 2^{-126} \\
&\approx 1.401 \times 10^{-45}
\end{aligned}$$

CHAPTER 3

Expression Evaluation

Contents

This chapter describes the ways in which an expression with floating-point operations can be evaluated in the PowerPC Numerics environment. The environment does not require that all floating-point operations be performed with a certain precision. Instead, it lets each implementation choose the most efficient precision to use. An implementation can dictate that all floating-point operations be performed with a given precision, or an implementation may define a method by which the best possible precision is chosen for each expression. This chapter describes the two methods that numeric implementations can use to choose a precision and compares the methods using several examples.

You should read this chapter to learn how PowerPC Numerics determines the precision of a floating-point expression.

About Expression Evaluation

The **evaluation format** of a floating-point operation is the data format used to evaluate the operation. An **expression evaluation method** is the means by which evaluation formats are determined. The IEEE standard does not cover expression evaluation methods, but the FPCE technical report does. Expression evaluation methods in PowerPC Numerics comply with the FPCE recommendations.

All PowerPC Numerics expression evaluation methods have a predefined minimum evaluation format, and they may or may not have widest-need evaluation. The **minimum evaluation format** is the narrowest evaluation format allowed for any operation. Any of the three floating-point data formats (single, double, or double-double) can be designated as the minimum evaluation format. **Widest-need evaluation** is a method used to determine the evaluation format for **complex expressions** (expressions with more than one floating-point operation). The following sections describe how expressions are evaluated without widest-need evaluation and with widest-need evaluation.

Evaluating Expressions Without Widest Need

Without widest-need evaluation, a complex expression is considered as a series of **simple expressions** (expressions with only one floating-point operation), and the evaluation format of each simple expression is determined separately. The evaluation format of a simple expression is either its **semantic type** (the widest format used for its operands) or the minimum evaluation format, whichever is wider. For example, consider the operation

s * *d*

where *s* is a single-format variable and *d* is a double-format variable. The operation's semantic type is double because double is the widest format used for an operand. If the minimum format is defined to be single, the operation is evaluated in double precision because double is wider than single. If the minimum format is double-double, double-double precision is used because double-double is wider than double. Evaluating this operation in double-double precision means that the values of both variables will be converted to double-double format before the multiplication is performed and that double-double format will be used for temporary storage of the result.

This expression evaluation method applies only to floating-point operations subject to the **usual arithmetic conversions** (automatic conversions performed in the C programming language). The following operations are subject to the usual arithmetic conversions:

- arithmetic operations
- comparison operations

The following operations are *not* subject to the usual arithmetic conversions:

- assignment
- assignment of actual function arguments to formal function parameters
- explicit conversions to different data types (for example, casts in C)

For example, consider the C expression

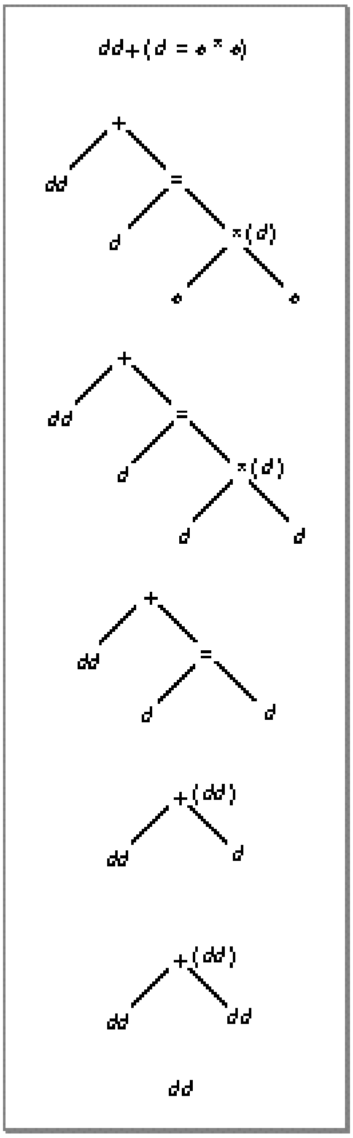
$$dd + (d = s * s)$$

where *dd* denotes a double-double format variable or number, *d* is double format, *s* is single format, and the minimum evaluation format is double. Without widest-need evaluation, this expression is treated as three simple expressions:

- $s * s$
- *d* assigned the result of $s * s$
- *dd* + the result of $d = s * s$

The semantic type of the first simple expression ($s * s$) is single, which is narrower than the minimum evaluation format, so it will be evaluated in double. The values of both of its operands are converted to double format and are then multiplied to produce a double result. The next simple expression is an assignment operation, which is not subject to the usual arithmetic conversions so the expression evaluation method does not apply. It produces a double format result also. Then, the last simple expression is considered. Its semantic type is double-double, so that will be the evaluation format. The result of the assignment is converted to double-double format, then added to the double-double variable. Figure 3-1 illustrates this process.

Figure 3-1 Evaluating complex expressions without widest need



Evaluating Expressions With Widest Need

Widest-need evaluation first looks at all of the operands of all of the subexpressions in a complex expression to determine the semantic type of the complex expression. As before, if the semantic type is wider than the minimum evaluation format, the semantic type is the evaluation format. If not, the minimum evaluation format is used. Only subexpressions with operations subject to the usual arithmetic conversions are considered when determining the evaluation format; operations such as assignment statements or casts are ignored.

After the evaluation format is determined, widest-need evaluation applies this format to the operands of the outermost operation in the expression using one of the following rules:

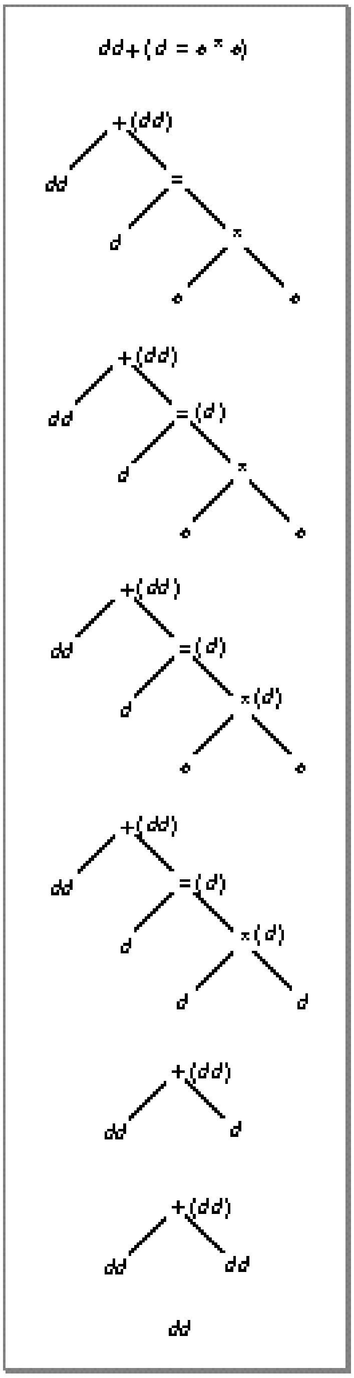
- If the operand is a floating-point variable or constant, it is converted to the evaluation format.
- If the operand is an operation subject to the usual arithmetic conversions (for example, arithmetic operations, comparison operations, and assignment of values to function parameters), its operands are converted to the evaluation format before the operation is performed.
- If the operand is an operation not subject to the usual arithmetic conversions (for example, an assignment operation, function call, or cast), its evaluation format is determined separately from the outer expression. After the operation has been performed, its result is converted to the evaluation format of the complex expression.

These three rules are applied repeatedly until the end of the expression is reached. For example, consider the C expression in Figure 3-2. Widest-need evaluation looks at this expression as the addition of a double-double variable to the result of another expression. To determine the evaluation format of this addition operation, widest need first looks at all of the variables and constants in the entire expression that are not part of a function call, cast, or assignment operation. There is only one variable that meets these requirements, and it is in double-double format. Therefore, double-double format is the evaluation format of the addition operation.

Now, widest-need evaluation can apply the addition operation's evaluation format to the rest of the expression using the three rules just given. Addition is an operation subject to the usual arithmetic conversions, and so its operands will be converted before the operation is performed. The first operand is a double-double variable, so it will be converted to the evaluation format immediately. (In this case, the variable already is in the evaluation format.) The second operand is an assignment operation. The assignment operation is not subject to arithmetic conversions, so it will be performed before any conversion takes place. This means that the evaluation format for the assignment operation must be determined. The operation's semantic type is double, so it will be performed in double precision.

As before, this double format must now be applied to the operands of the assignment. The first operand is already in double format. The second operand is a multiplication operation. Because multiplication is subject to the usual arithmetic conversions, its operands are converted before the operation is performed. Both of the multiplication operation's operands are single-format variables, so the values of these two variables are converted to double. The multiplication operation is calculated in double precision. Now the assignment can be performed, resulting in a double-format number. This result of the assignment statement is now the second operand of the addition operation. It is converted to double-double format, and then the addition is performed in double-double precision.

Figure 3-2 Evaluating complex expressions with widest need



Comparisons of Expression Evaluation Methods

You can think of the difference between using and not using widest-need evaluation as the way these two expression evaluation methods navigate the parse tree for a complex expression. Widest-need evaluation determines the evaluation format of the topmost expression first and enforces that format on all lower expressions. If a complex expression is evaluated without widest need, the evaluation format of the bottommost expression is determined first, and the results are converted to wider formats as wider formats are encountered working back up the tree.

Figure 3-3 shows how an expression is evaluated using both methods. In this example, *dd* is a double-double format constant or variable, *d* is double format, and *s* is single format. The minimum evaluation format is single. This expression makes a call to a function named `dfunc`, which takes a parameter of type `double` and returns a double value.

If this expression is evaluated without widest need, the evaluation format of the multiplication operation (*s* * *s*) is determined first without regard to the rest of the expression. Its semantic type is single, which is the same as the minimum evaluation format, so it is evaluated in single precision. Its result is then converted to double precision when it is passed to the function `dfunc`, which takes a double parameter. The function returns a double result. The next expression is the addition operation, which has a semantic type of double-double. The addition will be performed in double-double precision because double-double format is wider than the minimum evaluation format. The double-format return value from `dfunc` is converted to double-double, the addition is performed, and a double-double result is returned.

If this expression is evaluated with widest-need evaluation, the evaluation format of the addition operation is determined first. All of the variables in the expression that are not assigned to function parameters or not part of an assignment statement or cast are looked at to determine the evaluation format. In this expression, the two variables considered are the *dd* variable and the `dfunc` function call. Because *dd* is double-double format, the evaluation format of the addition operation is double-double. Now, the double-double format is applied down the parse tree to the operands of the addition operation. The first operand is already in double-double format. The second operand is a function call. As explained on page 3-6, function calls are not subject to the usual arithmetic conversions, so their evaluation formats are determined independently of the outer expression and their results are determined before any conversion takes place. The evaluation format for the assignment of values to `dfunc`'s parameters is double because `dfunc` takes a parameter of type `double`. The multiplication operation is an operand to this operation, so the multiplication is performed in double precision. The result of `dfunc` is returned in double format, then is converted to double-double format before the addition is performed.

Figure 3-3 Evaluating an expression with a function call

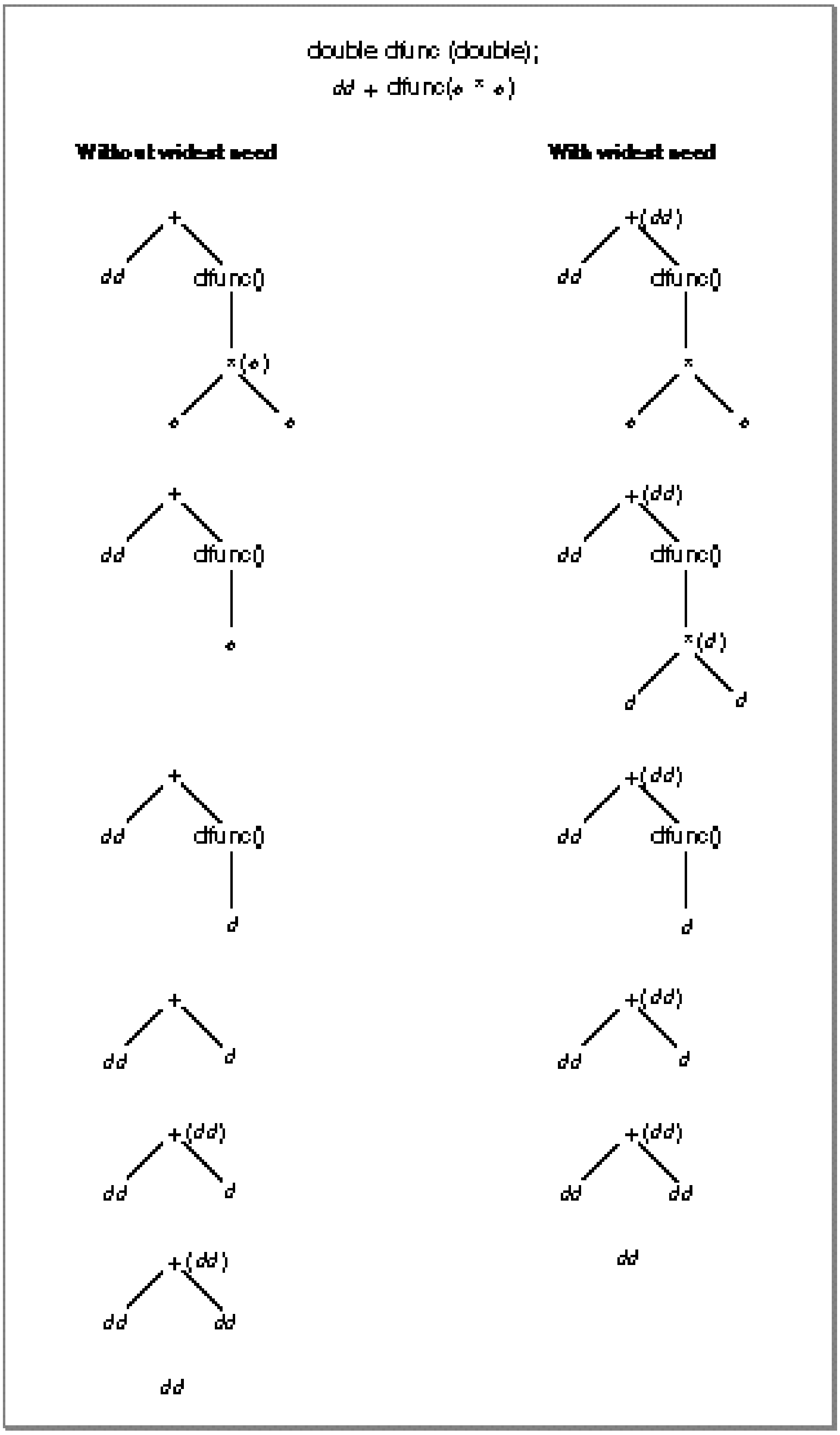
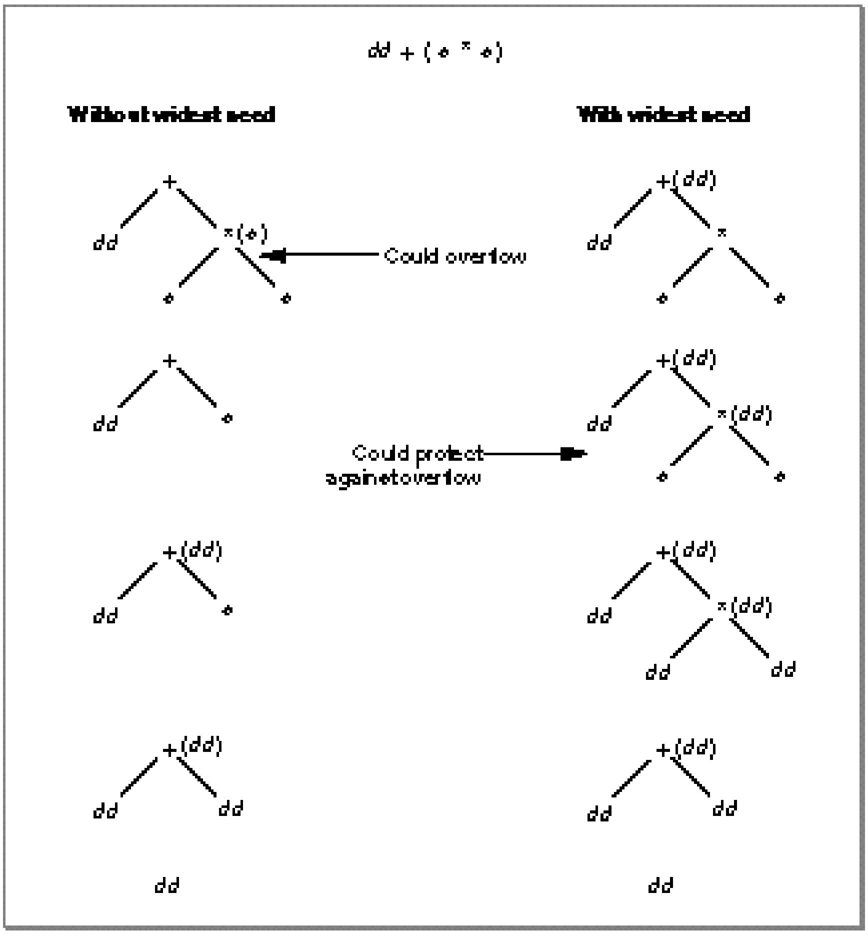


Figure 3-4 shows how widest-need evaluation protects against midexpression overflow and underflow better than expression evaluation methods that do not use widest need. In this example, *s* denotes a single format variable or number, *d* is double format, *dd* is double-double format, and the minimum evaluation format is single.

Figure 3-4 Evaluating an expression with arithmetic operations



Without widest-need evaluation, the expression in Figure 3-4 is considered as two separate simple expressions. The multiplication operation (*s* * *s*) is considered first. Its semantic type (single format) is the same as the minimum evaluation format, so the multiplication is performed in single precision. The semantic type of the addition operation is double-double, which is wider than the single minimum format. The addition operation is evaluated in double-double precision, so the value of its single-format operand is converted to double-double format before the result is calculated.

With widest-need evaluation, all of the operands in the complex expression are looked at first to determine the semantic type. The semantic type is double-double because of the double-double variable. This means that the multiplication of the two single-format variables is performed in double-double precision.

Suppose that the two single variables have the values 10^{38} and 10, respectively. Multiplying these two values produces 10^{39}. However, 10^{39} is out of the range of single format. If these numbers are multiplied in single precision (that is, if widest-need evaluation is not used), it will produce +∞ and a floating-point overflow exception. If the multiplication is evaluated in double-double precision (that is, if widest-need evaluation is used), the correct result is returned because 10^{39} is within the range of the double-double format.

CHAPTER 4

Environmental Controls

Contents

This chapter describes the parts of the floating-point environment that you can control. The IEEE standard specifies that users should be able to control the rounding direction, floating-point exceptions, and in some instances the rounding precision. PowerPC Numerics implementations provide utilities (called **environmental controls**) with which you can set, clear, and test the rounding direction and floating-point exception flags. (See Parts 2 and 3 for the exact names of functions and instructions that control the floating-point environment.) This chapter describes the four rounding direction modes and the five floating-point exception flags that you can set, clear, and test in PowerPC Numerics. You should read it to learn more about the floating-point environment.

Rounding Direction Modes

The available **rounding direction modes** are

- to nearest
- upward (toward +∞)
- downward (toward −∞)
- toward zero

The rounding direction affects all conversions, except conversions between decimal structures and decimal strings (described in Chapter 5, “Conversions”), and all arithmetic operations except remainder. All operations are calculated without regard to the range and precision of the data type in which the result is to be stored. That is, an operation first produces a result that is infinitely precise, or exact. If the destination data type cannot represent this number exactly, the result is rounded in the direction specified by the rounding mode.

The default rounding direction is to nearest. In this mode, floating-point expressions deliver the value nearest to the exact result that the destination data type can represent. If two representable values are equally close to the exact result, the expression delivers the one whose least significant bit is zero. Hence, halfway cases (for example, 1.5) round to even when the destination is an integer type or when the round-to-integer operation is used. If the magnitude of the exact result is greater than the data type’s largest value (by at least one half unit in the last place), then the Infinity with the corresponding sign is delivered.

The other rounding directions are upward, downward, and toward zero. When rounding upward, the result is the representable value (possibly +∞) closest to, and not less than, the exact result. When rounding downward, the result is the representable value (possibly −∞) closest to, and not greater than, the exact result. When rounding toward zero, the result is the representable value closest to, and not greater in magnitude than, the exact result. Toward-zero rounding truncates a number to an integer (when the destination is an integer type). Table 4-1 shows some values rounded to integers using different rounding modes.

Table 4-1 Examples of rounding to integer in different directions

Floating-point number	Rounded to nearest	Rounded toward 0	Rounded downward	Rounded upward
1.5	2	1	1	2
2.5	2	2	2	3
–2.2	–2	–2	–3	–2

Rounding Precision

A rounding precision mode specifies a precision to which all numeric operation results are rounded. For example, if the rounding precision mode were set to single, the results of all operations would be rounded to single precision until the rounding precision mode changed.

The IEEE standard requires rounding precision modes only on systems that always deliver results to double or extended format destinations. With a rounding precision mode set to a narrower format, such systems round to the precision of that format regardless of the destination. Thus, they can use rounding precision modes to emulate other systems that deliver results in narrower formats.

Because PowerPC Numerics delivers results in any of its three data formats, it does not support dynamic rounding precision modes. Instead, a PowerPC Numerics implementation may support static narrowing of rounding precision at translation time through pragmas, compiler options, or narrower expression evaluation methods.

Exception Flags

Floating-point exceptions are signaled with **exception flags**. When an application begins, all floating-point exception flags are cleared and the default rounding direction (to nearest) is in effect. This is the **default environment**. When an exception occurs, the appropriate exception flag is set, but the application continues normal operation. Floating-point exception flags merely indicate that a particular event has occurred; they do not change the flow of control for the application. An application can examine or set individual exception flags and can save and retrieve the entire environment (rounding direction and exception flags).

Note

The Exception Manager, described in the book *Inside Macintosh: PowerPC System Software*, does not report floating-point exceptions in the first version of the system software for PowerPC processor-based Macintosh computers. u

The numerics environment supports five exception flags:

- invalid operation (often called simply invalid)
- underflow
- overflow
- divide-by-zero
- inexact

These are discussed in the paragraphs that follow.

Invalid Operation

The **invalid exception** (or invalid-operation exception) occurs if an operand is invalid for the operation being performed. The result is a quiet NaN if the destination format is single, double, or double-double. The invalid conditions for the different operations are

Operation	Invalid condition
Addition or subtraction	Magnitude subtraction of Infinities, for example, $(+\infty) + (-\infty)$
Multiplication	$0 \times \infty$
Division	$0/0$ or ∞/∞
Remainder	*x* rem *y*, where *y* is 0 or *x* is infinite
Square root	A negative operand
Conversion	See Chapter 5, “Conversions”
Comparison	With predicates involving less than or greater than, but not unordered, when at least one operand is a NaN

In addition, any operation on a signaling NaN except the class and sign inquiries and, on some implementations, sign manipulations (absolute value and copysign) produce an invalid exception.

Underflow

The **underflow exception** occurs when a floating-point result is both tiny and inexact (and therefore is perhaps significantly less accurate than if there were no limit to the exponent range). A result is considered **tiny** if it must be represented as a denormalized number.

Overflow

The **overflow exception** occurs when the magnitude of a rounded floating-point result is greater than the largest finite number that the floating-point destination data format can represent. (Invalid, rather than overflow, flags the production of an out-of-range value for an integer destination type.)

Divide-by-Zero

The **divide-by-zero exception** occurs when a finite, nonzero number is divided by zero. It also occurs, in the more general case, when an operation on finite operands produces an exact infinite result; for example, logb (0) returns $-\infty$ and signals divide-by-zero. (Overflow, rather than divide-by-zero, flags the production of an inexact infinite result.)

Inexact

The **inexact exception** occurs if the rounded result of an operation is not identical to the exact (infinitely precise) result. Thus, an inexact exception always occurs when an overflow or underflow occurs. Valid operations on Infinities are always exact and therefore signal no exceptions. Invalid operations on Infinities are described at the beginning of this section.

CHAPTER 5

Conversions

Contents

This chapter describes how floating-point numbers can be converted in PowerPC Numerics. PowerPC Numerics can convert floating-point numbers to different data formats automatically or explicitly. For example, when a floating-point expression is evaluated, one or more of its operands might automatically be converted to a different data format. When a floating-point value is assigned to a variable, another automatic conversion might be necessary. You may also perform such conversions explicitly using the conversion utilities provided by your numeric implementation.

This chapter lists the supported numeric conversions and describes how each of these conversions is performed. You should read it to find out exactly how a floating-point value is converted to a different format. Chapter 3, “Expression Evaluation,” describes how PowerPC Numerics decides when operands must be converted during expression evaluation. Parts 2 and 3 describe the conversion utilities available to the users of different implementations.

About Conversions

The IEEE standard requires the following types of conversions:

- from floating-point formats to integer formats
- from integer formats to floating-point formats
- from floating-point values to integer values, with the result in a floating-point format
- between all supported floating-point formats
- between binary and decimal numbers

PowerPC Numerics supports all of these, as well as conversions between decimal formats.

Converting Floating-Point to Integer Formats

In the PowerPC Numerics environment, the following three types of floating-point to integer conversions are supported either directly by the programming languages or by library implementations:

- round to integer in current rounding direction (the required conversion, discussed in detail in Chapter 4, “Environmental Controls”)
- chop to integer (or round toward zero)
- add half to magnitude and chop

Although the IEEE standard specifies that conversions from floating-point to integer formats be rounded in the current rounding direction, high-level languages usually define their own methods. For example, the default method of converting from floating-point to integer formats in C is simply to discard the fractional part (**truncate**).

In general, when a language defines the rounding behavior for conversion to or from an integer, PowerPC Numerics languages conform.

Conversions from floating-point to integer formats raise the invalid floating-point exception flag in any of the following cases:

- The floating-point value is out of range for the integer type (for example, an attempt to convert a 64-bit integer value stored in the double data type to a 32-bit integer type).
- The floating-point value is a NaN.
- The floating-point value is an Infinity.

All floating-point to integer conversions that are in range but inexact (that is, the floating-point value was not an integer) raise the inexact floating-point exception flag, although this is not required by the IEEE standard.

Table 5-1 shows some examples of how floating-point values might be converted to a 32-bit integer format by rounding in the current rounding direction. Note that IEEE rounding in the default direction (to nearest) differs from most common rounding functions on halfway cases.

Table 5-1 Examples of floating-point to integer conversion

Floating-point number	Rounded to nearest	Rounded toward 0	Rounded downward	Rounded upward
1.5	2	1	1	2
2.5	2	2	2	3
–2.2	–2	–2	–3	–2
2,147,483,648.5	NaN	NaN	NaN	NaN

Rounding Floating-Point Numbers to Integers

PowerPC Numerics can also round floating-point numbers to integers and leave them stored in the same floating-point data format. These conversions may round in the current rounding direction, or they may explicitly round upward, downward, to the nearest value, or toward zero. These operations do not affect zeros, NaNs, or Infinities, because these three types of special values are already considered integers.

Converting Integers to Floating-Point Formats

When an integer is converted to a floating-point format whose precision is greater than or equal to the size of the integer format, the conversion is exact. When an integer is converted to a floating-point format whose precision is less than the size of the integer format, the integer is rounded in the current rounding direction. For example, because the single format has 24 bits in the significand, any integer requiring more than 24 bits of precision will not be converted to its exact value.

Converting Between Floating-Point Formats

PowerPC Numerics supports conversions between all three of its floating-point data formats. This section describes these conversions.

Converting Between Single and Double Formats

The PowerPC microprocessor directly supports the single and double formats and conversions between them. When a single format number is converted to a double format number, the conversion is exact.

When a double format number is converted to a single format number, it is rounded to the closest single value in the current rounding direction. The conversion might raise the exceptions shown in Table 5-2.

Table 5-2 Double to single conversion: Possible exceptions

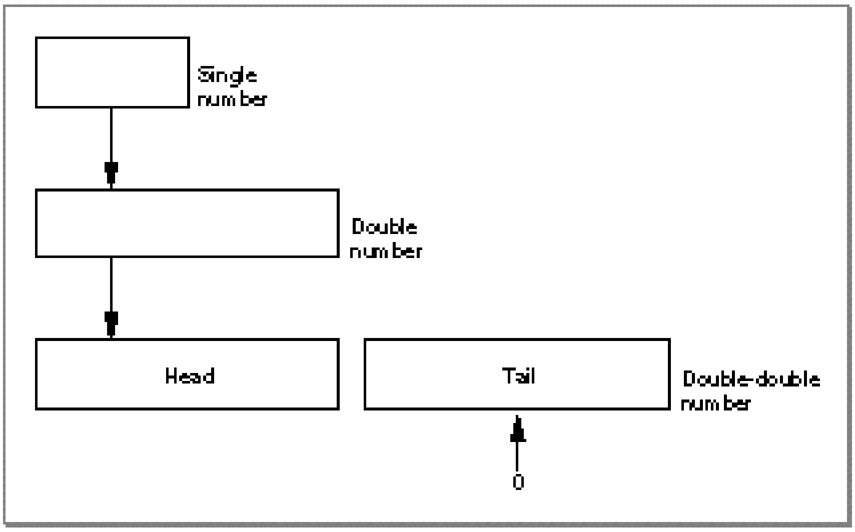
Exception	Raised when
Inexact	Significand requires > 24 bits of precision
Overflow	Exponent > 127
Underflow	Exponent < –126

Converting Between Single and Double-Double Formats

When a single format number is converted to a double-double format number, the result is exact. The following actions take place (as shown in Figure 5-1):

1. The single number is converted to double format.
2. The resulting double number is placed in the head of the double-double number.
3. The tail of the double-double number is set to 0.
4. The sign of the tail is set to the sign of the head.

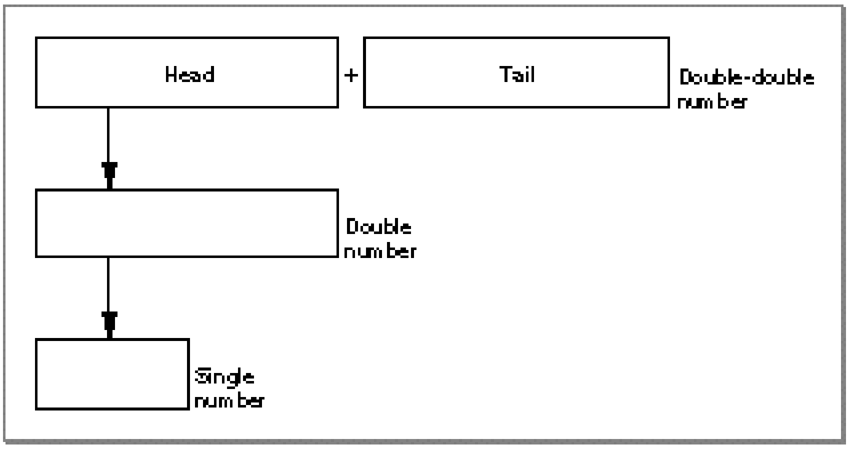
Figure 5-1 Single to double-double conversion



When a double-double number is converted to a single number, the following actions take place (as shown in Figure 5-2):

1. The head and tail of the double-double number are added together.
2. The sum is rounded to the closest single value in the current rounding direction.

Figure 5-2 Double-double to single conversion



The double-double to single conversion might raise the exceptions shown in Table 5-3.

Table 5-3 Double-double to single conversion: Possible exceptions

Exception	Raised when
Inexact	Significand requires > 24 bits of precision
Overflow	Exponent > 127
Underflow	Exponent < –126

Converting Between Double and Double-Double Formats

When a double format number is converted to a double-double format number, the result is exact. The following actions take place:

1. The double number is placed in the head of the double-double number.
2. The tail of the double-double number is set to 0.
3. The sign of the tail is set to the sign of the head.

When a double-double number is converted to a double number, the following actions take place:

1. The head and tail of the double-double number are added together.
2. The sum is rounded to the closest double value in the current rounding direction.

The conversion might raise the inexact exception if the significand requires more than 53 bits of precision.

Converting Between Binary and Decimal Numbers

PowerPC Numerics automatically converts between binary and decimal numbers, and some implementations allow you to perform such conversions manually. This section describes when conversions between binary and decimal numbers are performed and how they are performed.

Accuracy of Decimal-to-Binary Conversions

As explained in Chapter 1, “IEEE Standard Arithmetic,” some real numbers that can be represented exactly in decimal cannot be represented exactly as binary floating-point numbers. As a result, it is important that conversions between the two types of numbers be as accurate as possible. Given a rounding direction, for every decimal value there is a best—that is, correctly rounded—binary value for each binary format. Conversely, for any rounding direction, each binary value has a corresponding best decimal representation for a given decimal format. Ideally, binary-to-decimal conversions should obtain this best value to reduce accumulated errors.

Conversion functions in PowerPC Numerics meet or exceed the stringent error bounds specified by the IEEE standard. This means that even though in extreme cases the conversions do not deliver the correctly rounded results, the results they do deliver are very nearly as good as the correctly rounded results. (The IEEE standard does not specify error bounds for conversions involving values beyond the double format. See IEEE Standard 754-1985 for a more detailed description of error bounds.)

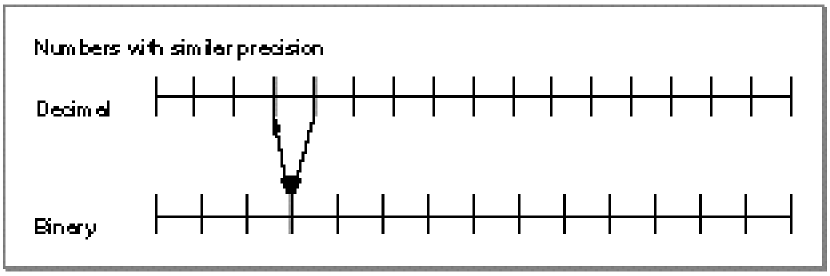
Automatic Conversions

Whenever a computer reads a decimal number into a binary format, it automatically converts the number to binary. Similarly, whenever a computer writes a binary number and a decimal format is specified for the output, it automatically converts the number from binary to decimal.

Suppose an application repeatedly reads and writes decimal data, meaning that it repeatedly converts values from decimal to binary and back. Such conversion cycles would occur, for example, in repeated execution of an application that updates a decimal file on a binary computer. Each time the application runs, it deliberately changes only a handful of values, but all the values get converted from decimal to binary and back again. Some computers use a conversion strategy that just drops extra digits; that is, it truncates the value. If the application were run on such a computer, the computer's rounding by truncation could cause severe downward drift. Using IEEE arithmetic with rounding to nearest, the values do not drift when you run the application repeatedly. That is, even though the conversions might change a few values the first time you run the program, there will be no further changes on subsequent conversions.

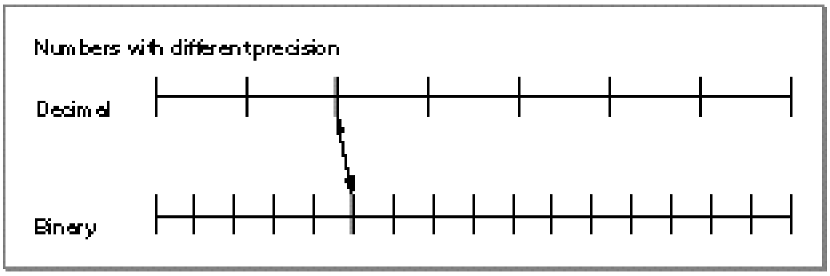
Figure 5-3 is a graphical model of such a conversion cycle with rounding to nearest, where the vertical marks represent decimal and binary computer numbers on the number line. The one-way arrow shows a decimal-to-binary conversion that does not get converted back to the original decimal value; the two-way arrow shows subsequent conversions returning the same value. In all cases, repeated conversions after the first give the same binary value; the error does not keep increasing.

Figure 5-3 Conversion cycle with first-time error



What’s more, if the binary format has enough extra precision beyond that of the decimal format, to-nearest rounding returns the original value the first time. The two-way arrow in Figure 5-4 shows a conversion cycle with different degrees of precision; here, the nearest decimal value to the binary result is always the original decimal value.

Figure 5-4 Conversion cycle with correct result



For the round-trip conversion from decimal to binary and back to decimal, the size of the decimal number you can start with and be sure that the round-trip produces the original value exactly depends on the binary data format. For single format, at most 6 decimal digits can be converted and return you the exact original value. For double format, 15 decimal digits, and for double-double format, 31 decimal digits.

You might also want to be sure conversions from binary numbers to decimal and back return the original value. For example, suppose your program writes out some stored values, and the output from this program is used as input to another program. You want to know how many decimal digits to print out to ensure that the conversion back to binary results in the original value. Again, the binary data format determines how many decimal digits are required for the conversion to return the original value. For single format, printing out 9 decimal digits insures an exact round trip; for double format, 17 decimal digits.

Note

These values bracket the ones given in Table 2-7 on page 2-16. u

Note that for the double-double format, because of its indefinite precision, there is no reasonable number of decimal digits you can print out to guarantee the conversion returns the original value. The number of decimal digits required varies with the difference between the head’s exponent value and the tail’s exponent. In the best case, the head’s exponent is exactly 54 greater than the tail’s exponent so that there is no gap between the head and the tail. In this case, 34 decimal digits are required to reproduce the original double-double value exactly. The worst case is when the tail is 0. No number of decimal digits is sufficient to provide an exact round trip when the tail is 0 (assuming an infinite exponent range).

Consider the case where a double-format number is converted to double-double format. For example, if you take 1.2 represented in double format and convert it to double-double format, the result (in hexadecimal) is

```
0x3FF33333 0x33333333 0x00000000 0x00000000
```

The first two hexadecimal numbers are stored in the head, and the last two are stored in the tail. Suppose you want to convert this double-double number to decimal. If you choose 34 decimal digits, the result is

```
1.199999999999999955591079014993738
```

This result is the closest 34-decimal digit approximation of the above double-double number. It is also the closest 34-decimal digit approximation of an infinitely precise binary value whose exponent is 0 and whose fractional part is represented by 13 sequences of “0011” followed by 52 binary zeros followed by some nonzero bits. When you convert this decimal value back to double-double format, PowerPC Numerics returns the closest double-double approximation of the infinitely precise value using all of the bits of precision available to it. That is, it will use all 53 bits in the head and 53 bits in the tail to store nonzero values and adjust the exponent of the tail accordingly. The result is

```
0x3FF33333 0x33333333 0xXXXYZZZZ 0xZZZZZZZZ
```

where XXX represents the sign and exponent of the tail, and Y represents the start of a nonzero value. Because the tail is always nonzero, this value is guaranteed to be not equal to the original double-double value.

Manual Conversions

A numeric implementation may provide functions that convert binary floating-point numbers to decimal and that convert decimal numbers to binary floating-point numbers. The decimal number can be input in one of two formats: as part of a decimal structure (described next) or as a character string. A numeric implementation also may provide a scanner for converting from decimal strings to decimal structures and a formatter for converting from decimal structures to decimal strings.

Converting Between Floating-Point and Decimal Structures

If the decimal number is part of a **decimal structure**, the structure contains

- a sign field
- an exponent field
- a significand field

For example, the file fp.h defines the following decimal structure for C:

```
typedef struct decimal
{
   char sgn;
   char unused;
   short exp;
   struct
   {
      unsigned char length;
      unsigned char text[SIGDIGLEN];
      unsigned chard unused;
   } sig;
} decimal;
```

The field sgn represents the sign, exp represents the exponent, and the structure sig represents the significand. The length field of the sig structure gives the length of the significand, and the character array text contains the significand. The decimal structure may either be input for a function that converts it to a binary floating-point number or output for a function that converts a binary floating-point number to this format.

IMPORTANT

When you create a decimal structure, you must set sig.length to the size of the string you place in sig.text. You cannot leave the length field undefined. s

Conversions from floating-point types to decimal structures also require a **decimal format structure** to specify how the decimal number should look. The decimal format structure contains the following information:

- whether the number should be in fixed or floating style
- if fixed style, the number of digits that should be to the right of the decimal point
- if floating style, the number of significant digits

For example, the file fp.h defines the decform structure for this purpose for the C programming language:

```
typedef struct decform
{
   char style;    /* FLOATDECIMAL OR FIXEDDECIMAL */
   char unused;
   short digits;
} decform;
```

Converting Between Floating-Point and Decimal Strings

Languages may provide routines to convert between numeric decimal strings and the numeric data formats. Note that conversions take place in the following cases:

- use of decimal constants in source code
- input of decimal strings (by procedures such as `read` in Pascal)
- calls to explicit routines

All conversions to decimal strings are controlled by a decimal formatting structure as described in the previous section.

CHAPTER 6

Numeric Operations and Functions

Contents

This chapter describes the operations (comparisons, arithmetic operations, and auxiliary and transcendental functions) that PowerPC Numerics allows you to perform on floating-point numbers. Numeric operations are evaluated as floating-point expressions; as such they are affected by, and might affect, the floating-point environment.

Read this chapter to find out what numeric operations are supported and how they work. For more information about how floating-point operations are evaluated in general, see Chapter 3, “Expression Evaluation.” For a description of the floating-point environment, see Chapter 4, “Environmental Controls.”

Comparisons

PowerPC Numerics supports the usual numeric comparisons: less than, less than or equal, greater than, greater than or equal, equal, and not equal (plus a few more described later). For real numbers, these comparisons behave according to the familiar ordering of real numbers.

Comparisons With NaNs and Infinities

Numeric comparisons handle NaNs and Infinities as well as real numbers. The usual trichotomy for real numbers is extended so that, for any numeric values *a* and *b*, exactly one of the following statements is true:

- $a < b$
- $a > b$
- $a = b$
- *a* and *b* are unordered

The following rule determines which statement is true: If *a* or *b* is a NaN, then *a* and *b* are unordered; otherwise, *a* is less than, equal to, or greater than *b* according to the ordering of the real numbers, with the understanding that

$$+0 = -0 \quad \text{and} \quad -\infty < \text{every real number} < +\infty$$

Comparison Operators

The meaning of high-level language relational operators is a natural extension of their old meaning based on trichotomy. For example, the C expression *x* <= *y* is true if *x* is less than *y* or if *x* equals *y*, and is false if *x* is greater than *y* or if *x* and *y* are unordered. Note that the numeric not-equal relation means less than, greater than, or unordered.

The FPCE technical report extends the usual set of C relational operators to a set of 14 comparisons, shown in Table 6-1.

Table 6-1 Comparison symbols

Symbol	Relation	Invalid if unordered?
<	Less than	Yes
>	Greater than	Yes
<=	Less than or equal to	Yes
>=	Greater than or equal to	Yes
==	Equal to	No
!=	Not equal to (unordered, less than, or greater than)	No
!<>=	Unordered	No
<>	Less than or greater than	Yes
<>=	Not unordered (less than, equal to, or greater than)	Yes
!<=	Not less than or equal to (unordered or greater than)	No
!<	Not less than (unordered, greater than, or equal to)	No
!>=	Not greater than or equal to (unordered or less than)	No
!>	Not greater than (unordered, less than, or equal to)	No
!<>	Unordered or equal	No

Some relational operators in high-level language comparisons contain the predicate less than or greater than, but not unordered. In C, those relational operators are <, <=, >, and >= (but not == and ! =). For those relations, comparisons signal invalid if the operands are unordered, that is, if either operand is a NaN. For the operators equal and nonequal, comparisons with NaN are not misleading; thus, when *x* or *y* is a NaN, the relation *x* == *y* is false, which is not misleading. Likewise, when *x* or *y* is a NaN, *x* != *y* returns true, again not misleading. On the other hand, when *x* or *y* is a NaN, *x* < *y* being false might tempt you to conclude that *x* ≥ *y*, so PowerPC Numerics signals invalid to help you avoid the pitfall. Table 6-1 shows the results of such comparisons in C.

The full 26 distinct comparison predicates of the IEEE standard may be obtained by logical negation of all of the operators except for == and !=, which never signal invalid. For example, (*x* < *y*) and !(*x* !< *y*) are logically equivalent for all possible values of *a* and *b*, but the former raises the invalid exception flag when *x* and *y* compare unordered while the latter does not.

A comparison with a signaling NaN as an operand always signals invalid, just as in arithmetic operations.

In addition to the comparison operators, there are also library functions that perform comparisons. See the section “Comparison Functions,” in Chapter 10, “Transcendental Functions.”

Arithmetic Operations

PowerPC Numerics provides the seven arithmetic operations required by the IEEE standard for its three data types, as shown for the language C in Table 6-2.

Table 6-2 Arithmetic operations in C

Operation	C symbol
Add	`+`
Subtract	`–`
Multiply	`*`
Divide	`/`
Square root	`sqrt`
Remainder	`remainder`
Round-to-integer	`rint`

The language processors for the PowerPC automatically use their chosen expression evaluation methods for the normal inline operators (+, –, *, /). All the arithmetic operations produce the best possible result: the mathematically exact result, coerced to the precision and range of the evaluation format. The coercions honor the user-selectable rounding direction and handle all exceptions according to the requirements of the IEEE standard (see Chapter 4, “Environmental Controls”).

Some of the arithmetic operations are implemented in software. These operations are declared to be type `double_t`, which is defined to be type `double`.

+

You can use the + symbol to add two real numbers.

```
x + y
```

`x`	Any floating-point number.
`y`	Any floating-point number.

DESCRIPTION

The + operator performs the standard addition of two floating-point numbers.

EXCEPTIONS

When *x* and *y* are both finite and nonzero, either the result of *x* + *y* is exact or it raises one of the following exceptions:

- inexact (if the result must be rounded or if an overflow or underflow occurs)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 6-3 shows the results when one of the operands of the addition operation is a zero, a NaN, or an Infinity. In this table, *x* is any floating-point number.

Table 6-3 Special cases for floating-point addition

Operation	Result	Exceptions raised
x + (+0)	*x*	None
x + (−0)	*x*	None
(−0) + (+0)	+0	None
(−0) + (−0)	−0	None
x + NaN	NaN	None*
x + (+∞)	+∞	None
x + (−∞)	−∞	None
+∞ + (−∞)	NaN	Invalid

* If the NaN is a signaling NaN, the invalid exception is raised.

–

You can use the – symbol to subtract one real number from another.

```
x – y
```

x	Any floating-point number.
y	Any floating-point number.

DESCRIPTION

The – operator performs the standard subtraction of two floating-point numbers.

EXCEPTIONS

When x and y are both finite and nonzero, either the result of $x - y$ is exact or it raises one of the following exceptions:

- inexact (if the result must be rounded or if an overflow or underflow occurs)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 6-4 shows the results when one of the operands of the subtraction operation is a zero, a NaN, or an Infinity. In this table, x is any floating-point number.

Table 6-4 Special cases for floating-point subtraction

Operation	Result	Exceptions raised
$x - (+0)$	x	None
$(+0) - x$	$-x$	None
$(+0) - (-0)$	$+0$	None
$x - (-0)$	x	None
$(-0) - x$	$-x$	None
$(-0) - (+0)$	-0	None
$(-0) - (-0)$	$+0$	None
x – NaN	NaN	None*
NaN – x	NaN	None*
$x - (+\infty)$	$-\infty$	None
$(+\infty) - x$	$+\infty$	None
$(+\infty) - (+\infty)$	NaN	Invalid
$x - (-\infty)$	$+\infty$	None
$(-\infty) - x$	$-\infty$	None
$(-\infty) - (-\infty)$	NaN	Invalid

* If the NaN is a signaling NaN, the invalid exception is raised.

*

You can use the * symbol to multiply two real numbers.

```
x * y
```

x	Any floating-point number.
y	Any floating-point number.

DESCRIPTION

The * operator performs the standard multiplication of two floating-point numbers $(x \times y)$.

EXCEPTIONS

When *x* and *y* are both finite and nonzero, either the result of *x* * *y* is exact or it raises one of the following exceptions:

- inexact (if the result of *x* * *y* must be rounded or if an overflow or underflow occurs)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 6-5 shows the results when one of the operands of the multiplication operation is a zero, a NaN, or an Infinity. In this table, *x* is a nonzero floating-point number.

Table 6-5 Special cases for floating-point multiplication

Operation	Result	Exceptions raised
x * +0	±0	None
x * −0	±0	None
±∞ * ±0	NaN	Invalid
x * NaN	NaN	None*
x * +∞	±∞	None
x * −∞	±∞	None
±0 * ±∞	NaN	Invalid

* If the NaN is a signaling NaN, the invalid exception is raised.

/

You can use the / symbol to divide one real number by another.

```
x / y
```

x	Any floating-point number.
y	Any floating-point number.

DESCRIPTION

The / operator performs the standard division of two floating-point numbers.

EXCEPTIONS

When *x* and *y* are both finite and nonzero, either the result of *x*/*y* is exact or it raises one of the following exceptions:

- inexact (if the result must be rounded or if an overflow or underflow occurs)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 6-6 shows the results when one of the operands of the division operation is a zero, a NaN, or an Infinity. In this table, *x* is any floating-point number.

Table 6-6 Special cases for floating-point division

Operation	Result	Exceptions raised
(+0)/*x*	±0	None
x/(+0)	±∞	Divide-by-zero
(−0)/*x*	±0	None
x/(−0)	±∞	Divide-by-zero
±0/±0	NaN	Invalid
x/NaN	NaN	None*
NaN/*x*	NaN	None*
x/(+∞)	±0	None

continued

Table 6-6 Special cases for floating-point division (continued)

Operation	Result	Exceptions raised
$(+\infty)/x$	±∞	None
x / $(-\infty)$	±0	None
$(-\infty)/x$	±∞	None
$(\pm\infty)/(\pm$	NaN	Invalid

[*] If the NaN is a signaling NaN, the invalid exception is raised.

sqrt

You can use the square root (`sqrt`) function to compute the square root of a real number.

```
double_t sqrt(double_t x);
```

`x` Any positive floating-point number.

DESCRIPTION

$$\text{sqrt}\ (x) = \sqrt{x}$$

EXCEPTIONS

When x is finite and nonzero, either the result of sqrt (x) is exact or it raises one of the following exceptions:

- inexact (if the result must be rounded)
- invalid (if x is negative)

SPECIAL CASES

Table 6-7 shows the results when the argument to the square root function is a zero, a NaN, or an Infinity, plus other special cases for the square root function. In this table, x is a finite, nonzero floating-point number.

Table 6-7 Special cases for floating-point square root

Operation	Result	Exceptions raised
sqrt (x) for $x < 0$	NaN	Invalid
sqrt (+0)	+0	None
sqrt (−0)	−0	None
sqrt (NaN)	NaN	None*
sqrt (+∞)	+∞	None
sqrt (−∞)	NaN	Invalid

* If the NaN is a signaling NaN, the invalid exception is raised.

remainder, remquo, and fmod

You can use the `remainder`, `remquo`, and `fmod` functions to perform the remainder operation recommended in the IEEE standard.

```
double_t remainder (double_t x, double_t y);
double_t remquo (double_t x, double_t y, int *quo);
double_t fmod (double_t x, double_t y);
```

`x`	Any floating-point number.
`y`	Any floating-point number.
`quo`	On return, the signed lowest seven bits (in the range of −127 to +127, inclusive) of the integer value closest to the quotient x/y. This partial quotient might be of use in certain argument reduction algorithms.

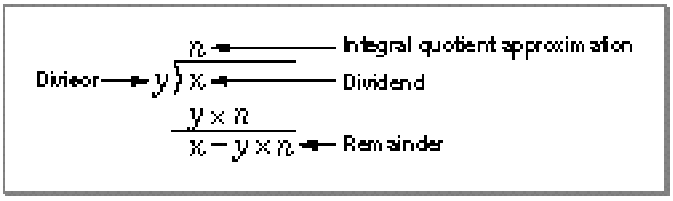
DESCRIPTION

The IEEE remainder (rem) operation returns the result of the following computation.

$r = x \text{ rem } y = x - y \times n$

where n is the integer nearest the exact value of the quotient x/y. This expression can be found even in the conventional integer-division algorithm, shown in Figure 6-1.

Figure 6-1 Integer-division algorithm



Whenever $|n - x/y| = 1/2$, n is even.

If the value of r is 0, the sign of r is that of x.

The rem operation is always exact.

The IEEE rem operation differs from other commonly used remainder and modulo operations. It returns a remainder of the smallest possible magnitude, and it always returns an exact remainder. Other remainder functions can be constructed from the IEEE remainder function by appropriately adding or subtracting *y*.

EXCEPTIONS

When *x* and *y* are finite, nonzero floating-point numbers in single or double format, the result of *x* rem *y* is exact.

SPECIAL CASES

Table 6-8 shows the results when one of the arguments to the rem operation is a zero, a NaN, or an Infinity. In this table, *x* is a finite, nonzero floating-point number.

Table 6-8 Special cases for floating-point remainder

Operation	Result	Exceptions raised
+0 rem *x*	+0	None
x rem (+0)	NaN	Invalid
−0 rem *x*	−0	None
x rem (−0)	NaN	Invalid
x rem NaN	NaN	None[*]
NaN rem *x*	NaN	None[*]
x rem +∞	*x*	None
+∞ rem *x*	NaN	Invalid
x rem −∞	*x*	None
−∞ rem *x*	NaN	Invalid

[*] If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = remainder(5, 3);    /* z = –1. */
/* 5 rem 3 = 5 – 3 × 2 = –1 because 1 < 5/3 < 2 and because
   5/3 = 1.66666... is closer to 2 than to 1, quo is taken to
   be 2.                                                    */

z = remainder(43.75, 2.5); /* z = –1.25. */
/* 43.75 rem 2.5 = 43.75 – 2.5 × 18 = –1.25 because
   17 < 43.75/2.5 < 18 and because 43.75/2.5 = 17.5 is
   equally close to both 17 and 18, quo is taken to be the
   even quotient, 18.                                       */

z = remainder(43.75, +INFINITY); /* z = 43.75 */
/* 43.75 rem ∞ = 43.75 – 0 × ∞ = 43.75 because 43.75 / ∞ = 0,
   quo is taken to be 0.                                    */
```

rint

You can use the round-to-integer operation (`rint`) to round a number to the nearest integer in the current rounding direction.

```
double_t rint(double_t x);
```

`x`	Any floating-point number.

DESCRIPTION

The `rint` function rounds its argument to an integer in the current rounding direction. The available rounding directions are upward, downward, to nearest (default), and toward zero. With the default rounding direction, if the argument is equally near two integers, the even integer is used as the result.

In each floating-point data type, all values of sufficiently great magnitude are integers. For example, in single format, all numbers whose magnitudes are at least 2^{23} are integers. This means that +∞ and −∞ are already integers and return exact results.

The `rint` function performs the round-to-integer arithmetic operation described in the IEEE standard. For other functions that perform rounding to integer, see Chapter 9, “Conversion Functions.”

EXCEPTIONS

When x is finite and nonzero, either the result of rint (x) is exact or it raises the following exception

- inexact (if x is not an integer)

SPECIAL CASES

Table 6-9 shows the results when the argument to the round-to-integer function is a zero, a NaN, or an Infinity.

Table 6-9 Special cases for floating-point round-to-integer

Operation	Result	Exceptions raised
rint (+0)	+0	None
rint (−0)	−0	None
rint (NaN)	NaN	None[*]
rint ($+\infty$)	$+\infty$	None
rint ($-\infty$)	$-\infty$	None

[*] If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

Table 6-10 shows some examples results of `rint`, given different rounding directions.

Table 6-10 Examples of `rint`

	Current rounding direction			
Example	To nearest	Toward 0	Downward	Upward
`rint(1.5)`	2	1	1	2
`rint(2.5)`	2	2	2	3
`rint(–2.2)`	–2	–2	–3	–2

Auxiliary Functions

The IEEE standard defines a number of recommended functions (called auxiliary functions) that are generally useful in numerical programming. The recommended functions supported by PowerPC Numerics are

- copysign (x, y) : copy the sign
- fabs (x) : absolute value
- logb (x) : binary exponent
- nan functions: NaN generators
- nextafter functions
- scalb (x) : binary scaling

The auxiliary functions are provided in the C library MathLib. For more information about these functions, see Part 2.

Transcendental Functions

PowerPC Numerics provides several basic mathematical functions in addition to the auxiliary functions recommended in the IEEE standard. These functions include

- logarithms
- exponentials
- two important financial functions
- trigonometric functions
- a random number generator
- error and gamma functions

For information about the transcendental functions supported, see Part 2.

PART TWO

The PowerPC Numerics C Implementation

This part describes the PowerPC Numerics implementation for the C programming language. The numeric implementation for the C language conforms to both IEEE standard 754, referred to in this book as the IEEE standard, and the recommendations in the FPCE technical report. As stated in Part 1, the FPCE report proposes a standard way of doing floating-point arithmetic for the C programming language. The IEEE standard specifies a standard for floating-point arithmetic for all computers regardless of the architecture or of any high-level language. The FPCE recommendations conform to the IEEE standard and standardize its implementation for the C programming language, so that if you write a program that uses FPCE features, it will compile with any FPCE-compliant compiler.

PowerPC Numerics in C is supported largely through a library called MathLib. This library contains macros, functions, and type definitions that provide conformance to the IEEE standard and the FPCE technical report. Some of the functions in the PowerPC Numerics library have two implementations: double precision and double-double precision. The double-double-precision implementation has the letter *l* appended to the name of the function and performs exactly the same as the double version. This book uses the double-precision implementation's name to mean both of these implementations.

This part describes the MathLib PowerPC Numerics library, its adherence to each piece of the PowerPC Numerics environment, and its additional features that conform to the FPCE technical report. For more information about the semantics of PowerPC Numerics, see Part 1. Read Part 2 if you are a programmer and you want to find out how to access the features described in Part 1 using the C language. You might also find Appendixes C, D, and E (in the back of this book) useful as reference material.

Numeric Data Types in C

Contents

This chapter describes the numeric data types available in C and shows how to determine the class and sign of values represented in numeric data types. As stated in Chapter 2, “Floating-Point Data Formats,” the PowerPC Numerics environment provides three numeric data formats: single (32 bits long), double (64 bits long), and double-double (two double formats combined, resulting in 128 bits). Each can represent normalized numbers, denormalized numbers, zeros, NaNs, and Infinities. See Chapter 2 for information about the numeric data formats and about how they represent values. Read this chapter to find out about the mapping of numeric formats to floating-point types in C, about the floating-point type declarations made in the PowerPC Numerics library (MathLib), and about the library utilities available that can determine the class of a floating-point value.

C Data Types

Table 7-1 shows how the PowerPC Numerics data formats map to the C floating-point variable types. This mapping follows the recommendations in the FPCE technical report.

Table 7-1 Names of data types

PowerPC Numerics format	**C type**
IEEE single	`float`
IEEE double	`double`
Double-double	`long double`

Efficient Type Declarations

MathLib contains two floating-point type definitions, `float_t` and `double_t` in the header `Types.h`. If you define a variable to be `float_t` or `double_t`, it means “use the most efficient floating-point format for this architecture.” Table 7-2 shows the definitions for `float_t` and `double_t` for both the PowerPC and 680x0 architecture.

Table 7-2 `float_t` and `double_t` types

Architecture	**`float_t` type**	**`double_t` type**
PowerPC	`float`	`double`
680x0	`long double`	`long double`

For the PowerPC architecture, the most natural format for computations is double, but the architecture allows computations in single precision as well. Therefore, for the PowerPC architecture, `float_t` is defined to be `float` (single precision) and `double_t` is defined to be `double`. The 680x0 architecture is based on an 80-bit double-extended format (known as extended) and performs all computations in this format regardless of the type of the operands. Therefore, `float_t` and `double_t` are both `long double` (extended precision) for the 680x0 architecture.

If you declare a variable to be type `double_t` and you compile the program as a PowerPC application, the variable is a `double`. If you recompile the same program as an 680x0 application, the variable is `long double`.

Inquiries: Class and Sign

MathLib provides macros you can use to determine the class and sign of a floating-point value. All of these macros return type `long int`. They are listed in Table 7-3.

Table 7-3 Class and sign inquiry macros

Macro	Value returned	Condition
`fpclassify(x)`	`FP_SNAN`	`x` is a signaling NaN
	`FP_QNAN`	`x` is a quiet NaN
	`FP_INFINITE`	`x` is −∞ or +∞
	`FP_ZERO`	`x` is +0 or −0
	`FP_NORMAL`	`x` is a normalized number
	`FP_SUBNORMAL`	`x` is a denormalized (subnormal) number
`isnormal(x)`	`TRUE`	`x` is a normalized number
`isfinite(x)`	`TRUE`	`x` is not −∞, +∞, or NaN
`isnan(x)`	`TRUE`	`x` is a NaN (quiet or signaling)
`signbit(x)`	`1`	The sign bit of `x` is 1 (`x` is negative)
	`0`	The sign bit of `x` is 0 (`x` is positive)

Creating Infinities and NaNs

MathLib defines the constants INFINITY and NAN, so that you can assign these values to variables in your program, and provides the following function that returns NaNs:

```
double nan  (const char *tagp);
```

The nan function returns a quiet NaN with a fraction field that is equal to the argument tagp. The argument tagp is a pointer to a string that will be copied into bits 8 through 15 of the NaN's fraction field. The string should specify a decimal number between 0 and 255. For example:

```
nan("32")
```

creates a NaN with code 32. If you supply a negative string, it is the same as supplying the string "0". If you supply a string greater than 255, it is the same as supplying the string "255". For a list of predefined NaN codes, see Chapter 2, "Floating-Point Data Formats."

Numeric Data Types Summary

This section summarizes the C constants, macros, functions, and type definitions associated with creating floating-point values or determining the class and sign of a floating-point value.

C Summary

Constants

```
#ifdef   powerc
#define          LONG_DOUBLE_SIZE     16
#elif    mc68881
#define          LONG_DOUBLE_SIZE     12
#else
#define          LONG_DOUBLE_SIZE     10
#endif      /* powerc */

#define          HUGE_VAL             __inf()
#define          INFINITY             __inf()
#define          NAN                  nan("255")
```

Class and Sign Inquiry Macros

```
#define  fpclassify  (x)   ((    sizeof (x) == LONG_DOUBLE_SIZE)     ?  \
                                 __fpclassify  (x)                   :  \
                                 (sizeof (x) == DOUBLE_SIZE)         ?  \
                                 __fpclassifyd (x)                   :  \
                                 __fpclassifyf (x))

#define  isnormal    (x)   ((    sizeof (x) == LONG_DOUBLE_SIZE)     ?  \
                                 __isnormal (x)                      :  \
                                 (sizeof (x) == DOUBLE_SIZE)         ?  \
                                 __isnormald (x)                     :  \
                                 __isnormalf (x))
```

```
#define  isfinite    (x)   ((    sizeof (x) == LONG_DOUBLE_SIZE)     ?  \
                                 __isfinite (x)                      :  \
                                 ( sizeof (x) == DOUBLE_SIZE)        ?  \
                                 __isfinited (x)                     :  \
                                 __isfinitef (x))

#define  isnan       (x)   ((    sizeof (x) == LONG_DOUBLE_SIZE)     ?  \
                                 __isnan (x)                         :  \
                                 (sizeof (x) == DOUBLE_SIZE)         ?  \
                                 __isnand (x)                        :  \
                                 __isnanf (x))

#define  signbit     (x)   ((    sizeof (x) == LONG_DOUBLE_SIZE)     ?  \
                                 __signbit (x)                       :  \
                                 (sizeof (x) == DOUBLE_SIZE)         ?  \
                                 __signbitd (x)                      :  \
                                 __signbitf (x))

enum NumberKind
{
   FP_SNAN = 0,                                  /* signaling NaN */
   FP_QNAN,                                      /* quiet NaN */
   FP_INFINITE,                                  /* + or – infinity */
   FP_ZERO,                                      /* + or – zero */
   FP_NORMAL,                                    /* all normal numbers */
   FP_SUBNORMAL                                  /* denormal numbers */
};
```

Data Types

```
#ifdef   powerpc
   typedef float float_t;
   typedef double double_t;
#else
   typedef long double float_t;
   typedef long double double_t;
#endif      /* powerpc */
```

Special Value Routines and Macros

Creating NaNs

```
double nan                  (const char *tagp);
```

CHAPTER 8

Environmental Control Functions

Contents

This chapter describes how to control the floating-point environment using functions defined in MathLib.

As described in Chapter 4, "Environmental Controls," the rounding direction and the exception flags are the parts of the environment that you can access. You can test and change the rounding direction, and you can test, set, and clear the exceptions flags. You may also save and restore both the rounding direction and exception flags together as a single entity. This chapter describes the functions that perform these tasks. For the definitions of rounding direction and exception flags, see Chapter 4.

Read this chapter to learn how to access and manipulate the floating-point environment in the C language. All of the environmental control function declarations appear in the file `fenv.h`.

IMPORTANT

If your compiler supports the environmental access switch described in Appendix D, "FPCE Recommendations for Compilers," the switch must be turned on in the program before you use any of the functions described in this chapter. s

Controlling the Rounding Direction

In MathLib, the following functions control the rounding direction:

`fegetround` Returns the current rounding direction.

`fesetround` Sets the rounding direction.

The four rounding direction modes are defined as the constants shown in Table 8-1.

Table 8-1 Rounding direction modes in MathLib

Rounding direction	Constant
To nearest	`FE_TONEAREST`
Toward zero	`FE_TOWARDZERO`
Upward	`FE_UPWARD`
Downward	`FE_DOWNWARD`

fegetround

You can use the `fegetround` function to save the current rounding direction.

```
int fegetround (void);
```

DESCRIPTION

The fegetround function returns an integer that specifies which rounding direction is currently being used. The integer it returns will be equal to one of the constants shown in Table 8-1. You can save the returned value in an integer variable to save the current rounding direction.

EXAMPLES

```
int rounddir;
double_t x, y, result;

rounddir = fegetround();      /* save rounding direction */

result = x + y;
if (rounddir == FE_TONEAREST)
   printf("The result was rounded to the nearest value.\n");
else if (rounddir == FE_UPWARD)
   printf("The result was rounded upward.\n");
else if (rounddir == FE_DOWNWARD)
   printf("The result was rounded downward.\n");
else if (rounddir == FE_TOWARDZERO)
   printf("The result was rounded toward zero.\n");
```

fesetround

You can use the fesetround function to change the rounding direction.

```
int fesetround (int round);
```

round	One of the four rounding direction constants (see Table 8-1).

DESCRIPTION

The fesetround function sets the rounding direction to the mode specified by its argument. If the value of round does not match any of the rounding direction constants, the function returns 0 and does not change the rounding direction.

By convention, if you change the rounding direction inside a function, first save the rounding direction of the calling function using fegetround and restore the saved direction at the end of the function. This way, the function does not affect the rounding direction of its caller. If the function is to be reentrant, then storage for the caller’s rounding direction must be local.

One reason to change the rounding direction would be to put bounds on errors (at least for the basic arithmetic operations and square root). Suppose you want to evaluate an expression such as

$$x = (a \times b + c \times d) / (f + g)$$

where *a*, *b*, *c*, *d*, *f*, and *g* are positive.

To make sure that the result is always larger than the exact value, you can change the expression such that all roundings cause errors in the same direction. The example that follows changes the rounding direction to compute an upper bound for the expression, and then restores the previous rounding.

EXAMPLES

```
double_t big_divide(void)
{
   double_t x_up, a, b, c, d, f, g;
   int r;               /* specifies rounding direction */

   r = fegetround();    /* save caller's rounding direction */
   fesetround(FE_DOWNWARD);
                        /* downward rounding for denominator */
   x_up = f + g;
   fesetround(FE_UPWARD);
                        /* upward rounding for expression */
   x_up = (a * b + c * d) / x_up;
   fesetround(r);
                        /* restore caller's rounding direction */
   return(x_up);
}
```

Controlling the Exception Flags

In MathLib, the following functions control the floating-point exception flags:

`feclearexcept`	Clears one or more exceptions.
`fegetexcept`	Saves one or more exception flags.
`feraiseexcept`	Raises one or more exceptions.
`fesetexcept`	Restores the state of one or more exception flags.
`fetestexcept`	Returns the value of one or more exception flags.

The five floating-point exception flags are defined as the constants shown in Table 8-2.

Table 8-2 Floating-point exception flags in MathLib

Exception	Constant
Inexact	FE_INEXACT
Divide-by-zero	FE_DIVBYZERO
Underflow	FE_UNDERFLOW
Overflow	FE_OVERFLOW
Invalid	FE_INVALID

MathLib also defines another constant, FE_ALL_EXCEPT, which is the logical OR of all five exceptions. Using FE_ALL_EXCEPT, you can manipulate all five floating-point exception flags as a single entity. The type fexcept_t also exists so that all the exception flags may be accessed at once.

feclearexcept

You can use the feclearexcept function to clear one or more floating-point exceptions.

```
void feclearexcept (int excepts);
```

excepts A mask indicating which floating-point exception flags should be cleared.

DESCRIPTION

The feclearexcept function clears the floating-point exceptions specified by its argument. The argument may be one of the constants in Table 8-2, two or more of these constants ORed together, or the constant FE_ALL_EXCEPT.

EXAMPLES

```
feclearexcept(FE_INEXACT);          /* clears the inexact flag */
feclearexcept(FE_INEXACT|FE_UNDERFLOW);
                     /* clears the inexact and underflow flags */
feclearexcept(FE_ALL_EXCEPT);             /* clears all flags */
```

fegetexcept

You can use the `fegetexcept` function to save the current value of one or more floating-point exception flags.

```
void fegetexcept (fexcept_t *flagp, int excepts);
```

`flagp` A pointer to where the exception flag values are to be stored.

`excepts` A mask indicating which exception flags to save.

DESCRIPTION

The `fegetexcept` function saves the values of the floating-point exception flags specified by the argument `excepts` to the area pointed to by the argument `flagp`. The `excepts` argument may be one of the constants in Table 8-2 on page 8-6, two or more of these constants ORed together, or the constant FE_ALL_EXCEPT.

EXAMPLES

```
fegetexcept(flagp, FE_INVALID);     /* saves the invalid flag */
fegetexcept(flagp, FE_INVALID|FE_OVERFLOW|FE_DIVBYZERO);
      /* saves the invalid, overflow, and divide-by-zero flags */
fegetexcept(flagp, FE_ALL_EXCEPT);        /* saves all flags */
```

feraiseexcept

You can use the `feraiseexcept` function to raise one or more floating-point exceptions.

```
void feraiseexcept (int excepts);
```

`excepts` A mask indicating which floating-point exception flags should be set.

DESCRIPTION

The `feraiseexcept` function sets the floating-point exception flags specified by its argument. The argument may be one of the constants in Table 8-2 on page 8-6, two or more of these constants ORed together, or the constant FE_ALL_EXCEPT.

EXAMPLES

```
feraiseexcept(FE_OVERFLOW);         /* sets the overflow flag */
feraiseexcept(FE_INEXACT|FE_UNDERFLOW);
                        /* sets the inexact and underflow flags */
feraiseexcept(FE_ALL_EXCEPT);                /* sets all flags */
```

fesetexcept

You can use the `fesetexcept` function to restore the values of the floating-point exception flags previously saved by a call to `fegetexcept`.

```
void fesetexcept (const fexcept_t *flagp, int excepts);
```

`flagp` A pointer to the values the floating-point exception flags should have.

`excepts` A mask indicating which exception flags should have their values changed.

DESCRIPTION

The `fesetexcept` function sets the floating-point exception flags indicated by the argument `excepts` to the values indicated by the argument `flagp`. The `excepts` argument may be one of the constants in Table 8-2 on page 8-6, two or more of these constants ORed together, or the constant FE_ALL_EXCEPT.

You must call `fegetexcept` before this function to set the `flagp` argument. This argument cannot be set in any other way.

EXAMPLES

```
fesetexcept(flagp, FE_INVALID);  /* restores the invalid flag */
fesetexcept(flagp, FE_INVALID|FE_OVERFLOW|FE_DIVBYZERO);
   /* restores the invalid, overflow, and divide-by-zero flags */
fesetexcept(flagp, FE_ALL_EXCEPT);        /* restores all flags */
```

fetestexcept

You can use the `fetestexcept` function to find out if one or more floating-point exceptions has occurred.

```
int fetestexcept (int excepts);
```

`excepts` A mask indicating which floating-point exception flags should be tested.

DESCRIPTION

The fetestexcept function tests the floating-point exception flags specified by its argument. The argument may be one of the constants in Table 8-2 on page 8-6, two or more of these constants ORed together, or the constant FE_ALL_EXCEPT.

If all exception flags being tested are clear, fetestexcept returns a 0. If one of the flags being tested is set, fetestexcept returns the constant associated with that flag. If more than one flag is set, fetestexcept returns the result of ORing their constants together. For example, if the inexact exception is set, fetestexcept returns FE_INEXACT. If both the inexact and overflow exceptions flags are set, fetestexcept returns FE_INEXACT | FE_OVERFLOW.

EXAMPLES

```
feraiseexcept(FE_DIVBYZERO|FE_OVERFLOW);
feclearexcept(FE_INEXACT|FE_UNDERFLOW|FE_INVALID);

/* Now the divide-by-zero and overflow flags are 1, and the
   rest of the flags are 0. */

i = fetestexcept(FE_INEXACT);
                              /* i = 0 because inexact is clear */
i = fetestexcept(FE_DIVBYZERO);
                              /* i = FE_DIVBYZERO */
i = fetestexcept(FE_UNDERFLOW);
                              /* i = 0 */
i = fetestexcept(FE_OVERFLOW);
                              /* i = FE_OVERFLOW */
i = fetestexcept(FE_ALL_EXCEPT);
                              /* i = FE_DIVBYZERO | FE_OVERFLOW */
i = fetestexcept(FE_INVALID | FE_DIVBYZERO);
                              /* i = FE_DIVBYZERO */
```

Accessing the Floating-Point Environment

MathLib defines four functions that access the entire floating-point environment:

fegetenv	Returns the current environment.
feholdexcept	Saves the previous environment and clears all exception flags.
fesetenv	Sets new environmental values.
feupdateenv	Restores a previously saved environment.

These functions take parameters of type fenv_t. Type fenv_t is the environment word type. In general, the environmental access functions either take a pointer to a variable of type fenv_t or accept the macro FE_DFL_ENV, which defines the default environment (default rounding direction and all exceptions cleared).

fegetenv

You can use the fegetenv function to save the current state of the floating-point environment.

```
void fegetenv (fenv_t *envp);
```

envp A pointer to an environment word that will store the current state of the environment upon the function's return.

DESCRIPTION

The fegetenv function saves the current state of the rounding direction modes and the floating-point exception flags in the object pointed to by its envp argument.

EXAMPLES

```
double_t func (double_t x, double_t y)
{
   fenv_t *env;

   x = x + y;        /* floating-point op; may raise exceptions */
   fegetenv(env);    /* save state of env after add */

   y = y * x;        /* floating-point op; may raise exceptions */
   .
   .
   .
}
```

feholdexcept

You can use the feholdexcept function to save the current floating-point environment and then clear all exception flags.

```
int feholdexcept (fenv_t *envp);
```

`envp` A pointer to an environment word where the environment should be saved.

DESCRIPTION

The `feholdexcept` function stores the current environment in the argument `envp` and clears the floating-point exception flags. Note that this function does not affect the rounding direction. It is the same as performing the following two calls:

```
fegetenv(envp);
feclearexcept(FE_ALL_EXCEPT);
```

Call `feholdexcept` at the beginning of a function so that the function can start with all exceptions cleared but not change the caller's environment. Use `feupdateenv` to restore the caller's environment at the end of the function. The `feupdateenv` function keeps any exceptions raised by the current function set while restoring the rest of the caller's environment. Thus, using `feholdexcept` and `feupdateenv` together preserves all raised floating-point exceptions while allowing new ones to be raised as well.

EXAMPLES

```
void subroutine(void)
{
   fenv_t *e;        /* local storage for environment */

   feholdexcept(e);  /* save caller's environment and
                        clear exceptions */

   /* subroutine's operations here */

  feupdateenv(e);    /* restore caller's environment */
}
```

fesetenv

You can use the `fesetenv` function to restore the floating-point environment.

```
void fesetenv (const fenv_t *envp);
```

`envp` A pointer to a word containing the value to which the environment should be set.

DESCRIPTION

The fesetenv function sets the floating-point environment to the value pointed to by its argument envp. The value of envp must come from a call to either fegetenv or feholdexcept, or it may be the constant FE_DFL_ENV, which specifies the default environment. In the default environment, all exception flags are clear and the rounding direction is set to the default.

EXAMPLES

```
double_t func (double_t x, double_t y)
{
   fenv_t *env;

   fesetenv(FE_DFL_ENV);      /* clear environment */

   x = x + y;        /* floating-point op; may raise exceptions */
   fegetenv(env);    /* save state of env after add */

   y = y * x;        /* floating-point op; may raise exceptions */
   fesetenv(env);
                  /* ignore environmental changes by times op */
   .
   .
   .
}
```

feupdateenv

You can use the feupdateenv function to restore the floating-point environment previously saved with feholdexcept.

```
void feupdateenv (const fenv_t *envp);
```

envp A pointer to the word containing the environment to be restored.

DESCRIPTION

The feupdateenv function, which takes a saved environment as argument, does the following:

1. It temporarily saves the exception flags (raised by the current function).
2. It restores the environment received as an argument.
3. It signals the temporarily saved exceptions.

The feupdateenv function facilitates writing subroutines that appear to their callers to be **atomic operations** (such as addition, square root, and others). Atomic operations pass extra information back to their callers by signaling exceptions; however, they hide internal exceptions, which might be irrelevant or misleading. Thus, exceptions signaled between the feholdexcept and feupdateenv functions are hidden from the calling function unless the exceptions remain raised when the feupdateenv procedure is called.

EXAMPLES

```
/* NumFcn signals underflow if its result is denormalized,
overflow if its result is INFINITY, and inexact always, but hides
spurious exceptions occurring from internal computations. */

long double NumFcn(void)
{
   fenv_t e;                    /* local environment storage */
   enum NumKind c;              /* for class inquiry */
   fexcept_t * flagp;
   long double result;

   feholdexcept(&e);            /* save caller’s environment and
                                  clear exceptions */

      /* internal computation */

   c = fpclassify(result);      /* class inquiry */

   feclearexcept(FE_ALL_EXCEPT); /* clear all exceptions */
   feraiseexcept(FE_INEXACT);   /* signal inexact */

   if (c == FP_INFINITE)
      feraiseexcept(FE_OVERFLOW);
   else if (c == FP_SUBNORMAL)
      feraiseexcept(FE_UNDERFLOW);

   feupdateenv(&e);
   /* restore caller’s environment, and then signal
      exceptions raised by NumFcn                  */

   return(result);
}
```

Environmental Controls Summary

This section summarizes the C constants, macros, functions, and type definitions associated with controlling the floating-point environment.

C Summary

Constants

Rounding Direction Modes

```
#define  FE_TONEAREST      0x00000000
#define  FE_TOWARDZERO     0x00000001
#define  FE_UPWARD         0x00000002
#define  FE_DOWNWARD       0x00000003
```

Floating-Point Exception Flags

```
#define  FE_INEXACT        0x02000000     /* inexact */
#define  FE_DIVBYZERO      0x04000000     /* divide-by-zero */
#define  FE_UNDERFLOW      0x08000000     /* underflow */
#define  FE_OVERFLOW       0x10000000     /* overflow */
#define  FE_INVALID        0x20000000     /* invalid */

#define  FE_ALL_EXCEPT     (  FE_INEXACT | FE_DIVBYZERO | FE_UNDERFLOW | \
                              FE_OVERFLOW | FE_INVALID )

#define  FE_DFL_ENV        &_FE_DFL_ENV   /* pointer to default environment*/
```

Data Types

```
typedef     long int    fenv_t;

typedef     long int    fexcept_t;
```

Environment Access Routines

Controlling Rounding Direction

```
int fegetround              (void);
int fesetround              (int round);
```

Controlling the Exception Flags

```
void feclearexcept          (int excepts);
void fegetexcept            (fexcept_t *flagp, int excepts);
void feraiseexcept          (int excepts);
void fesetexcept            (const fexcept_t *flagp, int excepts);
int fetestexcept            (int excepts);
```

Accessing the Floating-Point Environment

```
void fegetenv               (fenv_t *envp);
int feholdexcept            (fenv_t *envp);
void fesetenv               (const fenv_t *envp);
void feupdateenv            (const fenv_t *envp);
```

CHAPTER 9

Conversion Functions

Contents

This chapter describes how you can perform the conversions required by the IEEE standard using MathLib C functions. For each type of conversion, this chapter lists the functions you can use to perform that conversion. It shows the declarations of these functions, describes what they do, describes when they raise floating-point exceptions, and gives examples of how to use them. For a description of the conversions required by the IEEE standard and the details of how each conversion is performed in PowerPC Numerics, see Chapter 5, “Conversions.” All of the conversion function declarations appear in the file `fp.h`.

Converting Floating-Point to Integer Formats

In C, the default method of converting floating-point numbers to integers is to simply discard the fractional part (truncate). MathLib provides two functions that convert floating-point numbers to integers using methods other than the default C method and that return the integers in integer types.

rinttol (*x*)	Returns the nearest integer to *x* in the current rounding direction as an integer type.
roundtol (*x*)	Adds 1/2 to the magnitude of *x*, chops to an integer, and returns the value as an integer type.

rinttol

You can use the `rinttol` function to round a real number to the nearest integer in the current rounding direction.

```
long int rinttol (double_t x);
```

`x`	Any floating-point number.

DESCRIPTION

The `rinttol` function rounds its argument to the nearest integer in the current rounding direction and places the result in a `long int` type. The available rounding directions are upward, downward, to nearest, and toward zero.

The `rinttol` function provides the floating-point to integer conversion as described in the IEEE standard. It differs from `rint` (described on page 6-13) in that it returns the value in an integer type; `rint` returns the value in a floating-point type.

EXCEPTIONS

When *x* is finite and nonzero, either the result of rinttol (*x*) is exact or it raises one of the following exceptions:

- inexact (if *x* is not an integer)
- invalid (if the integer result is outside the range of the `long int` type)

SPECIAL CASES

Table 9-1 shows the results when the argument to the `rinttol` function is a zero, a NaN, or an Infinity.

Table 9-1 Special cases for the `rinttol` function

Operation	Result	Exceptions raised
rinttol (+0)	+0	None
rinttol (−0)	−0	None
rinttol (NaN)	Undefined	None*
rinttol (+∞)	Undefined	Invalid
rinttol (−∞)	Undefined	Invalid

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = rinttol(+INFINITY);/* z = unspecified value for all rounding
                         directions because +INFINITY exceeds the
                         range of long int. The invalid exception
                         is raised. */
z = rinttol(300.1);    /* z = 301 if rounding direction is upward
                         else z = 300. The inexact exception is
                         raised.*/
z = rinttol(–300.1);   /* z = –301 if rounding direction is
                         downward else z = –300. The inexact
                         exception is raised. */
```

roundtol

You can use the `roundtol` function to round a real number to the nearest integer value by adding 1/2 to the magnitude and truncating.

```
long int roundtol (double_t x);
```

`x` Any floating-point number.

DESCRIPTION

The `roundtol` function adds 1/2 to the magnitude of its argument and chops to integer, returning the answer in `long int` type.

The result is returned in an integer data type. (The return type is the difference between `roundtol` and the `round` function described on page 9-10.)

This function is not affected by the current rounding direction. Notice that the `roundtol` function rounds halfway cases (1.5, 2.5, and so on) away from 0. With the default rounding direction, `rinttol` (described on page 9-3) rounds halfway cases to the even integer.

EXCEPTIONS

When *x* is finite and nonzero, either the result of roundtol (*x*) is exact or it raises one of the following exceptions:

- inexact (if *x* is not an integer)
- invalid (if the integer result is outside the range of the `long int` type)

SPECIAL CASES

Table 9-2 shows the results when the argument to the `roundtol` function is a zero, a NaN, or an Infinity.

Table 9-2 Special cases for the `roundtol` function

Operation	Result	Exceptions raised
roundtol (+0)	+0	None
roundtol (−0)	−0	None
roundtol (NaN)	Undefined	None*
roundtol (+ ∞)	Undefined	Invalid
roundtol (−∞)	Undefined	Invalid

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = roundtol(+INFINITY);   /* z = an unspecified value because
                              +∞ is outside of the range of long
                              int. */
z = roundtol(0.5);         /* z = 1 because |0.5| + 0.5 = 1.0. The
                              inexact exception is raised. */
z = roundtol(–0.9);        /* z = –1 because |–0.9| + 0.5 = 1.4.
                              The inexact exception is raised. */
```

Rounding Floating-Point Numbers to Integers

MathLib provides six functions that convert floating-point numbers to integers and return the integer in the floating-point type. The first is the `rint` function, which performs the round-to-integer operation as described in Chapter 6, "Numeric Operations and Functions." The other functions either round in a specific direction or perform a variation of the `rint` operation.

ceil (*x*)	Returns the nearest integer not less than *x*.
floor (*x*)	Returns the nearest integer not greater than *x*.
nearbyint (*x*)	Returns the nearest integer to *x* in the current rounding direction.
round (*x*)	Adds 1/2 to the magnitude of *x* and chops to an integer.
trunc (*x*)	Truncates the fractional part of *x*.

ceil

You can use the `ceil` function to round a real number upward to the nearest integer value.

```
double_t ceil (double_t x);
```

`x`	Any floating-point number.

DESCRIPTION

The `ceil` function rounds its argument upward. This is an ANSI standard C library function. The result is returned in a floating-point data type.

This function is the same as performing the following code sequence:

```
r = fegetround();       /* save current rounding direction */
fesetround(FE_UPWARD);  /* round upward */
rint(x);                /* round to integer */
fesetround(r);          /* restore rounding direction */
```

EXCEPTIONS

When x is finite and nonzero, the result of ceil (x) is exact.

SPECIAL CASES

Table 9-3 shows the results when the argument to the `ceil` function is a zero, a NaN, or an Infinity.

Table 9-3 Special cases for the `ceil` function

Operation	Result	Exceptions raised
ceil (+0)	+0	None
ceil (−0)	−0	None
ceil (NaN)	NaN	None[*]
ceil (+∞)	+∞	None
ceil (−∞)	−∞	None

[*] If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = ceil(+INFINITY); /* z = +INFINITY because +INFINITY is already
                        an integer value by definition. */
z = ceil(300.1);     /* z = 301.0 */
z = ceil(–300.1);    /* z = –300.0 */
```

floor

You can use the `floor` function to round a real number downward to the next integer value.

```
double_t floor (double_t x);
```

`x` Any floating-point number.

DESCRIPTION

The floor function rounds its argument downward. This is an ANSI standard C library function. The result is returned in a floating-point data type.

This function is the same as performing the following code sequence:

```
r = fegetround();           /* save current rounding direction */
fesetround(FE_DOWNWARD);    /* round downward */
rint(x);                    /* round to integer */
fesetround(r);              /* restore rounding direction */
```

EXCEPTIONS

When x is finite and nonzero, the result of floor (x) is exact.

SPECIAL CASES

Table 9-4 shows the results when the argument to the floor function is a zero, a NaN, or an Infinity.

Table 9-4 Special cases for the floor function

Operation	Result	Exceptions raised
floor (+0)	+0	None
floor (−0)	−0	None
floor (NaN)	NaN	None*
floor (+∞)	+∞	None
floor (−∞)	−∞	None

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = floor(+INFINITY);   /* z = +INFINITY because +∞ is already an
                           integer value by definition. */
z = floor(300.1);       /* z = 300.0 */
z = floor(–300.1);      /* z = –301.0 */
```

nearbyint

You can use the nearbyint function to round a real number to the nearest integer in the current rounding direction.

```
double_t nearbyint (double_t x);
```

x   Any floating-point number.

DESCRIPTION

The nearbyint function rounds its argument to the nearest integer in the current rounding direction. The available rounding directions are upward, downward, to nearest, and toward zero.

The nearbyint function provides the floating-point to integer conversion described in the IEEE Standard 854. It differs from rint (described on page 6-13) only in that it does not raise the inexact flag when the argument is not already an integer.

EXCEPTIONS

When x is finite and nonzero, the result of nearbyint (x) is exact.

SPECIAL CASES

Table 9-5 shows the results when the argument to the nearbyint function is a zero, a NaN, or an Infinity.

Table 9-5 Special cases for the nearbyint function

Operation	Result	Exceptions raised
nearbyint (+0)	+0	None
nearbyint (−0)	−0	None
nearbyint (NaN)	NaN	None*
nearbyint (+∞)	+∞	None
nearbyint (−∞)	−∞	None

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = nearbyint(+INFINITY);  /* z = +INFINITY for all rounding
                              directions. */
z = nearbyint(300.1);      /* z = 301.0 if rounding direction is
                              upward, else z = 300.0. */
z = nearbyint(–300.1);     /* z = –301.0 if rounding direction is
                              downward, else z = –300.0. */
```

round

You can use the `round` function to round a real number to the integer value obtained by adding 1/2 to the magnitude and truncating.

```
double_t round (double_t x);
```

`x` Any floating-point number.

DESCRIPTION

The `round` function adds 1/2 to the magnitude of its argument and chops to integer. The result is returned in a floating-point data type.

This function is not affected by the current rounding direction. Notice that the `round` function rounds halfway cases (1.5, 2.5, and so on) away from 0. With the default rounding direction, `rint` (described on page 6-13) rounds halfway cases to the even integer.

EXCEPTIONS

When *x* is finite and nonzero, either the result of round (*x*) is exact or it raises the following exception:

- inexact (if *x* is not an integer value)

SPECIAL CASES

Table 9-6 shows the results when the argument to the `round` function is a zero, a NaN, or an Infinity.

Table 9-6 Special cases for the `round` function

Operation	Result	Exceptions raised
round (+0)	+0	None
round (−0)	−0	None
round (NaN)	NaN	None[*]
round (+∞)	+∞	None
round (−∞)	−∞	None

[*] If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = round(+INFINITY);   /* z = +INFINITY because +∞ is already an
                           integer value by definition. */
z = round(0.5);         /* z = 1.0 because |0.5| + 0.5 = 1.0. The
                           inexact exception is raised. */
z = round(–0.9);        /* z = –1.0 because |–0.9| + 0.5 = 1.4.
                           The inexact exception is raised. */
```

trunc

You can use the `trunc` function to truncate the fractional part of a real number so that just the integer part remains.

```
double_t trunc (double_t x);
```

`x` Any floating-point number.

DESCRIPTION

The `trunc` function chops off the fractional part of its argument. This is an ANSI standard C library function.

This function is the same as performing the code sequence:

```
r = fegetround();          /* save current rounding direction */
fesetround(FE_TOWARDZERO); /* round toward zero */
rint(x);                   /* round to integer */
fesetround(r);             /* restore rounding direction */
```

EXCEPTIONS

When *x* is finite and nonzero, the result of trunc (*x*) is exact.

SPECIAL CASES

Table 9-7 shows the results when the argument to the `trunc` function is a zero, a NaN, or an Infinity.

Table 9-7 Special cases for the `trunc` function

Operation	Result	Exceptions raised
trunc (+0)	+0	None
trunc (−0)	−0	None
trunc (NaN)	NaN	None[*]
trunc (+∞)	+∞	None
trunc (−∞)	−∞	None

[*] If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = trunc(+INFINITY);   /* z = +INFINITY because +∞ is already an
                           integer value by definition. */
z = trunc(300.1);       /* z = 300.0 */
z = trunc(–300.1);      /* z = –300.0 */
```

Converting Integers to Floating-Point Formats

In the C programming language, conversions from integers stored in an integer format to floating-point formats are automatic when you assign an integer to a floating-point variable.

```
double d;
int x = 1;
d = x;      /* value 1 automatically converted to double format */
```

Converting Between Floating-Point Formats

In the C programming language, conversions between floating-point formats are automatic when you assign a floating-point number of one type to a variable of another type.

```
float f = 0.0f;         /* single format */
double d = 1.1;
long double ld;         /* double-double format */

f = d;      /* double 1.1 converted to single format */
ld = f;     /* single 1.1 converted to double-double format */
d = ld;     /* double-double 1.1 converted to double format */
```

Converting Between Binary and Decimal Numbers

MathLib provides two functions that let you manually convert between binary and decimal formats.

dec2num	Converts a decimal number to a binary number.
num2dec	Converts a binary number to a decimal number.

Conversions between binary floating-point numbers and decimal numbers use structures of type decimal. The decimal structure is defined in the header file fp.h as

```
struct decimal
{
   char sgn;
   char unused;
   short exp;
   struct
   {
      unsigned char length;
      unsigned char text[SIGDIGLEN];
      unsigned char unused;
   } sig;
} decimal;
```

sgn The sign of the number (0 is positive, 1 is negative).

exp The exponent of the number. The exponent is expressed as a power of 10.

`sig` The significand. String `sig.text` contains the significand as a decimal integer in the form of a string, that is, with the string length in the zeroth byte (`sig.length`) and the initial character of the string in the first byte (`sig.text[0]` to `sig.text[SIGDIGLEN – 1]`).

The value represented is

$$(-1)^{\mathtt{sgn}} \times \mathtt{sig} \times 10^{\mathtt{exp}}$$

For example, if `sgn` equals 1, `exp` equals −3, and `sig` equals “85” (string length `sig.length` equals 2, not shown), then the number represented is −0.085.

Note
The maximum length of the string `sig` is implementation dependent.
The limit is 36 characters. Also, the representations of 0 and 1 in the 16-bit word `sgn` are implementation dependent. u

Conversions from binary to decimal use a decimal format structure to specify how the number should look in decimal. The `decform` structure is defined in the header file `fp.h` as

```
struct decform
{
   char style; /* FLOATDECIMAL or FIXEDDECIMAL */
   char unused;
   short digits;
} decform;
```

`style` The style of output. This field equals 0 (`FLOATDECIMAL`) for floating and 1 (`FIXEDDECIMAL`) for fixed.

`digits` The number of significant digits for the floating style and the number of digits to the right of the decimal point for the fixed style. (The value of `digits` may be negative if the style is fixed.)

Note
Formatting details, such as the representations of 0 and 1 in the 16-bit `style` word, are implementation dependent. u

If the `style` field of the `decform` structure equals 0 (in C, `f.style == FLOATDECIMAL`), the output is formatted in floating style, with the `digits` field specifying the number of significant digits required. Output in floating style is represented in the following format; Table 9-8 defines its components.

`[- |     ]`*m*`[.`*nnn*`]e[+ | -]`*dddd*

Table 9-8 Format of decimal output string in floating style

Component	Description
Minus sign (–) or space	Minus sign if `sgn` is 1; space if `sgn` is 0
m	Single digit, 0 only if value represented is 0
Point (.)	Present if `digits` > 1
nnn	String of digits; present if `digits` > 1
`e`	The letter *e*
Plus sign (+) or minus sign (–)	Plus sign if `exp` ≥ 0; minus sign if `exp` < 0.
dddd	One to four exponent digits

If the `style` field of the `decform` structure equals 1 (in C, `f.style == FIXEDDECIMAL`), the output is formatted in fixed style, with the `digits` field specifying the number of digits to follow the decimal point. All output in fixed style is represented in the following format; Table 9-9 defines its components.

`[-]`*mmm*`[.`*nnn*`]`

Table 9-9 Format of decimal output string in fixed style

Component	Description
Minus sign (–)	Present if `sgn` = 1
mmm	String of digits; at least one digit but no superfluous leading zeros
Point (.)	Present if `digits` > 0
nnn	String of digits of length equal to `digits`; present if `digits` > 0

Note that if `sgn` equals 0, then floating-style output begins with a space but fixed-style output does not.

Double-double values being converted to decimal strings are first rounded to 113 bits (if they in fact span more than that number of bits in their significands) and then converted to the decimal string of the desired length.

dec2num

You can use the dec2num function to convert a decimal number to a binary floating-point number.

```
float dec2f (const decimal *d);
double_t dec2num (const decimal *d);
long double dec2numl (const decimal *d);
short int dec2s (const decimal *d);
long int dec2l (const decimal *d);
```

d The decimal structure to be converted. See page 9-13 for the definition of decimal structure.

DESCRIPTION

The dec2num function converts a decimal number in a decimal structure to a double format floating-point number. Conversions from the decimal structure type handle any sig string of length 36 or less (with an implicit decimal point at the right end).

There are three versions of this function that convert to a floating-point type: dec2f converts the decimal number to the float type, dec2num converts to the double type, and dec2numl converts to the long double type. The other two versions of this function, dec2s and dec2l, convert to the short and long integer types, respectively.

IMPORTANT

When you create a decimal structure, you must set sig.length to the size of the string you place in sig.text. You cannot leave the length field undefined. s

You can use the numeric formatter (str2dec) before using this function to convert a decimal string to a decimal structure suitable for input to the dec2num function.

EXCEPTIONS

When the sig string is longer than 36 characters, the result is undefined.

SPECIAL CASES

The following special cases apply:

- If sig.text[0] is “0” (zero), the decimal structure is converted to zero. For example, a decimal structure with sig = “0913” is converted to zero.

- If sig.text[0] is “N”, the decimal structure is converted to a NaN. The succeeding characters of sig are interpreted as a hexadecimal representation of the result’s significand: if fewer than four characters follow the N, then they are right aligned in the high-order 15 bits of the field *f* illustrated in the section “Formats” in Chapter 2, “Floating-Point Data Formats”; if four or more characters follow the N, then they are left aligned in the result’s significand.
- If sig.text[0] is “I”, the decimal structure is converted to an Infinity.

EXAMPLES

```
decimal d;
double_t result;

d.sgn = 0;
d.exp = 3;
d.sig.length = 3;
d.sig.text[0] = '2';
d.sig.text[1] = '0';
d.sig.text[2] = '8';
result = dec2num(&d);      /* result = 208,000 stored in double
                              format */
```

num2dec

You can use the num2dec function to convert a binary floating-point number to a decimal number.

```
void num2dec (const decform *f, double_t x, decimal *d);
void num2decl (const decform *f, long double x, decimal *d);
```

f — A decform structure that describes how the number should look in decimal. See page 9-14 for a description of the decform structure.

x — The floating-point number to be converted.

d — Upon return, a pointer to the decimal structure containing the number. See page 9-13 for a description of the decimal structure.

DESCRIPTION

The num2dec function converts a floating-point number to a decimal number. The decimal number is contained in a decimal structure. Each conversion to a decimal structure d is controlled by a decform structure f. All implementations allow 36 digits to be returned in the sig field of the decimal structure. The implied decimal point is at the right end of sig, with exp set accordingly.

After using the `num2dec` function, you can use the `dec2str` function to convert the `decimal` structure to a character string.

IMPORTANT

Use the same decimal format structure settings for `dec2str` as you used for `num2dec`; otherwise, the results are unspecified. s

EXCEPTIONS

When the number of digits specified in a `decform` structure exceeds an implementation maximum (which is 36), the result is undefined.

A number might be too large to represent in a chosen fixed style. For instance, if the implementation's maximum length for `sig` is 36, then 10^{35} (which requires 33 digits to the left of the point in fixed-style representations) is too large for a fixed-style representation specifying more than two digits to the right of the point. If a number is too large for a chosen fixed style, then (depending on the numeric implementation) one of two results is returned: an implementation might return the most significant digits of the number in `sig` and set `exp` so that the `decimal` structure contains a valid floating-style approximation of the number; alternatively, an implementation might simply set `sig` to the string "?". Note that in any implementation, the following test determines whether a nonzero finite number is too large for the chosen fixed style.

```
decimal d;
decform f;
int too_big; /* Boolean */

too_big = (-d.exp != f.digits) || (d.sig.text[0] == "?");
```

For fixed-point formatting, PowerPC Numerics treats a negative value for `digits` as a specification for rounding to the left of the decimal; for example, `digits` = –2 means to round to hundreds. For floating-point formatting, a negative value for `digits` gives unspecified results.

SPECIAL CASES

- For zeros, the character "0" is placed in `sig.text[0]`.
- For NaNs, The character "N" is placed in `sig.text[0]`. The character "N" might be followed by a hexadecimal representation of the input significand. The third and fourth hexadecimal digits following the "N" give the NaN code. For example, "N4021000000000000" has NaN code 0x21.
- For Infinities, the character "I" is placed in `sig.text[0]`.

In all three of these cases, `exp` is undefined.

EXAMPLES

```
decimal d;
decform f;
double_t fp_num = 1.000007;

f.style = FLOATDECIMAL;    /* floating-point format */
f.digits = 7;              /* seven significant digits */
num2dec(&f, fp_num, &d);   /* d now contains 1.000007 expressed
                              in decimal structure */
```

Converting Between Decimal Formats

MathLib provides a scanner for converting from decimal strings to `decimal` structures and a formatter for converting from `decimal` structures to decimal strings.

`dec2str`	Converts `decimal` structures to decimal strings. The PowerPC Numerics formatter.
`str2dec`	Converts decimal strings to `decimal` structures. The PowerPC Numerics scanner.

dec2str

You can use the `dec2str` function to convert a number in a `decimal` structure to a decimal string.

```
void dec2str (const decform *f, const decimal *d, char *s);
```

`f` A `decform` structure that describes how the number should look in decimal. See page 9-14 for a description of the `decform` structure.

`d` The `decimal` structure to be converted. See page 9-13 for the definition of the `decimal` structure.

`s` On return, a string representing the number in decimal.

DESCRIPTION

The `dec2str` function is the PowerPC Numerics formatter. It takes a number from a decimal structure and converts it to a string. You can use the `num2dec` function to convert a binary floating-point number to a `decimal` structure appropriate for input to the `dec2str` function.

IMPORTANT

Use the same decimal format structure settings for `dec2str` as you used for `num2dec`; otherwise, results are unspecified. s

The numeric formatter is controlled by a `decform` structure `f`. With floating style, numbers formatted using the same value for `f.digits` have aligning decimal points and *e*'s. To ensure that numbers have the same width also, pad the exponent-digits field with spaces to a width of 4. For example, if `f.digits` = 12, then pad 12 + 8 – length(`s`) spaces on the right of the result string `s`. The value 8 accounts for the sign, point, letter *e*, exponent sign, and four exponent digits. Note that this scheme gives the correct field width for NaNs and Infinities too.

With fixed style, numbers formatted using the same value for `f.digits` have aligning decimal points if enough leading spaces are added to the result string `s` to attain a fixed width, which must be no narrower than the widest `s`.

IMPORTANT

When you create a `decimal` structure, you must set `sig.length` to the size of the string you place in `sig.text`. You cannot leave the `length` field undefined. s

EXCEPTIONS

The formatter is always exact and signals no exceptions.

SPECIAL CASES

For fixed-point formatting, `dec2str` treats a negative value for `digits` as a specification for rounding to the left of the decimal; for example, `digits` = –2 means to round to hundreds. For floating-point formatting, a values for `digits` less than 1 are treated as 1.

NaNs are formatted as `NAN`; Infinities are formatted as `INF`. A leading sign or space is included according to the style convention.

The formatter never returns fewer significant digits than are contained in `sig`. However, if the `decform` structure calls for more significant digits than are contained in `sig`, then the formatter pads with zeros as needed.

If more than 80 characters are required to honor `digits`, then the formatter returns the string "?".

EXAMPLES

Suppose you have an accounting program that computes exact values using binary numbers of pennies and prints outputs in dollars and cents. If you simply divide the number of pennies by 100 to get dollars, you incur errors because hundredths are not exact in binary. One way to print out exact values in dollars and cents is to convert the number of pennies to a `decimal` structure, perform the division by adjusting the exponent, and print the result, as shown in Listing 9-1.

Listing 9-1 Accounting program

```
#include <fp.h>

   decform     df;
   double      pennies;    /* This is the input value */
   decimal     dpennies;   /* decimal value for pennies */
   char *      dollars;    /* string to print as $$$.¢¢ */

{
   df.style = FIXEDDECIMAL;
   df.digits = 0;    /* start with 0 digits after decimal point */

   num2dec(&df, pennies, &dpennies);         /* decimal pennies */
   dpennies.exp = dpennies.exp – 2;          /* divide by 100 */

   df.digits = 2;       /* request 2 digits after decimal point */
   dec2str(&df, &dpennies, dollars);
   /* dollar string to print */
}
```

str2dec

You can use the `str2dec` function to convert a decimal string to a `decimal` structure.

```
void str2dec (const char *s, short *ix, decimal *d, short *vp);
```

`s` The character string containing the number to be converted.

`ix` On entry, the starting position in the string. On return, one greater than the position of the last character in the string that was parsed if the entire string was not converted successfully.

`d` On return, a pointer to the `decimal` structure containing the decimal number. See page 9-13 for a description of the `decimal` structure.

`vp` On return, a Boolean argument indicating the success of the function. If the entire string was parsed, `vp` is true. If part of the string was parsed, `vp` is false and `ix` indicates where the function stopped parsing.

DESCRIPTION

The `str2dec` function is the PowerPC Numerics scanner, which is designed for use both with fixed strings and with strings being received interactively character by character. The scanner parses the longest possible numeric substring; if no numeric substring is recognized, then the value of `ix` remains unchanged.

To convert floating-point strings embedded in text, parse to the beginning of a floating-point string ([+ | –] *digit*) and pass the current scan location as the index into the text. The conversion routine will return the value scanned and a new value of the index for continued parsing.

You might need to distinguish those numeric ASCII strings that represent values of an integer format. You can do this by scanning the source, looking for integer syntax. You can handle integers yourself and send to the numeric scanner any strings with floating-point syntax (that is, containing a period (.), an E, or an e). You might also want to pass along to the scanner any strings that cause integer overflow.

EXCEPTIONS

The scanner signals no exceptions. It faithfully converts all values within range that are representable in the `decimal` structure format.

SPECIAL CASES

To convert a zero, NaN, or Infinity, use one of the following as input:

```
–0   +0   0   –INF   Inf   NAN   –NaN()   nan
```

EXAMPLES

Listing 9-2 shows an example of how to scan decimal strings input into an application and then convert the strings to binary floating-point numbers using MathLib functions. Table 9-10 shows some sample inputs to the loop shown in Listing 9-2 and the results after each string has been converted to a decimal structure using `str2dec`.

Listing 9-2 Scanning algorithm

```
s = "";        /* initialize string */

/* loop until string is not a valid prefix*/
do
  {

   /* code to get next character and append to string goes here */

   /* scan string */
  ix = 0;
  str2dec(s, &ix, &d, &vp);
  }
while (vp = false);

/* convert from decimal to numeric-format result */
result = dec2num(d);
```

Table 9-10 Examples of conversions to `decimal` structures

	Index			
Input string	In	Out	**Output value**	**Valid prefix**
12	0	2	12	True
12E	0	2	12	True
12E–	0	2	12	True
12E–3	0	5	12E–3	True
12E–X	0	2	12	False
12E–3X	0	5	12E–3	False
x12E–3	1	6	12E–3	True
IN	0	0	NAN	True
INF	0	3	INF	True

Conversions Summary

This section summarizes the C constants, macros, functions, and type definitions associated with converting floating-point values.

C Summary

Constants

```
#define      SIGDIGLEN      36            /* significant decimal digits */
#define      DECSTROUTLEN   80            /* max length for dec2str output */
#define      FLOATDECIMAL   ((char)(0))
#define      FIXEDDECIMAL   ((char)(1))
```

Data Types

```
struct decimal
{
   char sgn;                              /* sign 0 for +, 1 for – */
   char unused;
   short exp;                             /* decimal exponent */
   struct
   {
      unsigned char length;
      unsigned char text[SIGDIGLEN];      /* significant digits */
      unsigned char unused;
   } sig;
};
typedef struct decimal decimal;

struct decform
{
   char style;                         /* FLOATDECIMAL or FIXEDDECIMAL */
   char unused;
   short digits;
};
typedef struct decform decform;
```

Conversion Routines

Converting Floating-Point Formats to Integer Formats

```
long int rinttol            (double_t x);
long int roundtol           (double_t x);
```

Rounding Floating-Point Numbers to Integers

```
double_t ceil               (double_t x);
double_t floor              (double_t x);
double_t nearbyint          (double_t x);
double_t round              (double_t x);
double_t trunc              (double_t x);
```

Converting Decimal Numbers to Binary Numbers

```
float dec2f                 (const decimal *d);
double_t dec2num            (const decimal *d);
long double dec2numl        (const decimal *d);
short int dec2s             (const decimal *d);
long int dec2l              (const decimal *d);
```

Converting Binary Numbers to Decimal Numbers

```
void num2dec                (const decform *f, double_t x, decimal *d);
void num2decl               (const decform *f, long double x, decimal *d);
```

Converting Between Decimal Formats

```
void dec2str                (const decform *f, const decimal *d, char *s);
void str2dec                (const char *s, short *ix, decimal *d,
                             short *vp);
```

CHAPTER 10

Transcendental Functions

Contents

This chapter describes how to use the transcendental and auxiliary functions declared in MathLib. This chapter describes the following types of functions:

- comparison
- sign manipulation
- exponential
- logarithmic
- trigonometric
- hyperbolic
- financialv
- error and gamma

It shows the declarations of these functions, describes what they do, describes when they raise floating-point exceptions, and gives examples of how to use them. For functions that manipulate the floating-point environment, see Chapter 8, “Environmental Control Functions.” For functions that perform conversions, see Chapter 9, “Conversion Functions.” For basic arithmetic and comparison operations, see Chapter 6, “Numeric Operations and Functions.”

Some transcendental functions have two implementations: double precision and double-double precision. The double-double-precision implementation has the letter *l* appended to the name of the function and performs exactly the same as the double version. This book uses the double-precision implementation’s name to mean both of these implementations. All of the transcendental function declarations appear in the file `fp.h`.

Comparison Functions

MathLib provides four functions that perform comparisons between two floating-point arguments:

fdim (x, y)	Returns the positive difference $x - y$ or 0.
fmax (x, y)	Returns the maximum of x or y.
fmin (x, y)	Returns the minimum of x or y.
relation (x, y)	Returns the relationship between x and y.

These functions take advantage of the rule from the IEEE standard that every value besides NaNs have an order:

$$-\infty < \text{all negative real numbers} < -0 = +0 < \text{all positive real numbers} < +\infty$$

These functions also make special cases of NaNs so that they raise no floating-point exceptions.

fdim

You can use the `fdim` function to determine the positive difference between two real numbers.

```
double_t fdim (double_t x, double_t y);
```

`x`	Any floating-point number.
`y`	Any floating-point number.

DESCRIPTION

The `fdim` function returns the positive difference between its two arguments.

$$\begin{aligned} \text{fdim}\ (x, y) &= x - y && \text{if } x > y \\ \text{fdim}\ (x, y) &= +0 && \text{if } x \le y \end{aligned}$$

EXCEPTIONS

When x and y are finite and nonzero and $x > y$, either the result of fdim (x, y) is exact or it raises one of the following exceptions:

- inexact (if the result of $x - y$ must be rounded)
- overflow (if the result of $x - y$ is outside the range of the data type)
- underflow (if the result of $x - y$ is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 10-1 shows the results when one of the arguments to the `fdim` function is a zero, a NaN, or an Infinity. In this table, x and y are finite, nonzero floating-point numbers.

Table 10-1 Special cases for the `fdim` function

Operation	Result	Exceptions raised
fdim $(+0, y)$	+0	None
fdim $(x, +0)$	x	None
fdim $(-0, y)$	+0	None
fdim $(x, -0)$	x	None
fdim (NaN, y)	NaN*	None†
fdim (x, NaN)	NaN	None†

Table 10-1 Special cases for the `fdim` function (continued)

Operation	Result	Exceptions raised
fdim $(+\infty, y)$	$+\infty$	None
fdim $(x, +\infty)$	$+0$	None
fdim $(-\infty, y)$	$+0$	None
fdim $(x, -\infty)$	$+\infty$	None

* If both arguments are NaN, the first NaN is returned.
† If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = fdim(+INFINITY, 300);  /* z = +∞ – 300 = +INFINITY because
                              +∞ > 300 */
z = fdim(300, +INFINITY);  /* z = +0 because 300 ≤ +∞ */
```

fmax

You can use the `fmax` function to find out which is the larger of two real numbers.

```
double_t fmax (double_t x, double_t y);
```

`x`	Any floating-point number.
`y`	Any floating-point number.

DESCRIPTION

The `fmax` function determines the larger of its two arguments.

$$\text{fmax}(x, y) = x \quad \text{if } x \geq y$$
$$\text{fmax}(x, y) = y \quad \text{if } x < y$$

If one of the arguments is a NaN, the other argument is returned.

EXCEPTIONS

When x and y are finite and nonzero, the result of fmax (x, y) is exact.

SPECIAL CASES

Table 10-2 shows the results when one of the arguments to the `fmax` function is a zero, a NaN, or an Infinity. In this table, *x* is a finite, nonzero floating-point number. (Note that the order of operands for this function does not matter.)

Table 10-2 Special cases for the `fmax` function

Operation	Result	Exceptions raised
fmax (+0, *x*)	x if x > 0	None
	+0 if x < 0	
fmax (−0, *x*)	x if x > 0	None
	–0 if x < 0	
fmax (±0, ±0)	+0	None
fmax (NaN, *x*)	*x*[*]	None[†]
fmax (+∞, *x*)	+∞	None
fmax (−∞, *x*)	x	None

[*] If both arguments are NaNs, the first NaN is returned.
[†] If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = fmax(–INFINITY, –300,000);   /* z = –300,000 because any
                                    integer is greater than –∞ */
z = fmax(NAN, –300,000);   /* z = –300,000 by definition of the
                              function fmax. */
```

fmin

You can use the `fmin` function to determine which is the smaller of two real numbers.

```
double_t fmin (double_t x, double_t y);
```

`x`	Any floating-point number.
`y`	Any floating-point number.

DESCRIPTION

The `fmin` function determines the lesser of its two arguments.

$$\begin{aligned} \mathrm{fmin}\,(x, y) &= x \quad \text{if } x \le y \\ \mathrm{fmin}\,(x, y) &= y \quad \text{if } y < x \end{aligned}$$

If one of the arguments is a NaN, the other argument is returned.

EXCEPTIONS

When x and y are finite and nonzero, the result of $\mathrm{fmin}\,(x, y)$ is exact.

SPECIAL CASES

Table 10-3 shows the results when one of the arguments to the `fmin` function is a zero, a NaN, or an Infinity. In this table, x is a finite, nonzero floating-point number. (Note that the order of operands for this function does not matter.)

Table 10-3 Special cases for the `fmin` function

Operation	Result	Exceptions raised
$\mathrm{fmin}\,(+0, x)$	x if $x < 0$	None
	$+0$ if $x > 0$	
$\mathrm{fmin}\,(-0, x)$	x if $x < 0$	None
	$+0$ if $x > 0$	
$\mathrm{fmin}\,(\pm 0, \pm 0)$	$+0$	None
$\mathrm{fmin}\,(\mathrm{NaN}, x)$	x*	None†
$\mathrm{fmin}\,(+\infty, x)$	x	None
$\mathrm{fmin}\,(-\infty, x)$	$-\infty$	None

* If both arguments are NaNs, the first NaN is returned.

† If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = fmin(-INFINITY, -300,000);   /* z = -INFINITY because -∞ is
                                    smaller than any integer. */
z = fmin(NAN, -300,000);   /* z = -300,000 by definition of the
                              function fmin. */
```

relation

You can use the `relation` function to determine the relationship (less than, greater than, equal, or unordered) between two real numbers.

```
relop relation (double_t x, double_t y);
```

`x`	Any floating-point number.
`y`	Any floating-point number.

DESCRIPTION

The `relation` function returns the relationship between its two arguments.

The `relation` function is type `relop`, which is an enumerated type. This function returns one of the following values:

if `x` > `y`	`GREATERTHAN`
if `x` < `y`	`LESSTHAN`
if `x` = `y`	`EQUALTO`
if `x` or `y` is a NaN	`UNORDERED`

Programs can use the result of this function in expressions to test for combinations not supported by the comparison operators, such as “less than or unordered.”

EXCEPTIONS

When x and y are finite and nonzero, the result of relation (x, y) is exact.

SPECIAL CASES

Table 10-4 shows the results when one of the arguments to the `relation` function is a zero, a NaN, or an Infinity. In this table, x and y are finite, nonzero floating-point numbers.

Table 10-4 Special cases for the `relation` function

Operation	Result	Exceptions raised
relation $(+0, y)$	< if $y > 0$	None
	> if $y < 0$	None
relation $(x, +0)$	> if $x > 0$	None
	< if $x < 0$	None

Table 10-4 Special cases for the `relation` function (continued)

Operation	Result	Exceptions raised
relation (−0, *y*)	< if *y* > 0	None
	> if *y* < 0	None
relation (*x*, −0)	> if *x* > 0	None
	< if *x* < 0	None
relation (+0, −0)	=	None
relation (NaN, *y*)	Unordered	None*
relation (*x*, NaN)	Unordered	None*
relation (+∞, *y*)	>	None
relation (*x*, +∞)	<	None
relation (+∞, +∞)	=	None
relation (−∞, *y*)	<	None
relation (*x*, −∞)	>	None
relation (−∞, −∞)	=	None

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
r = relation(x, y);
if ((r == LESSTHAN) || (r == UNORDERED))
   printf("No, y is not greater than x.\n");
```

Sign Manipulation Functions

MathLib provides two functions that manipulate the sign bit of the floating-point value:

copysign (*x*, *y*)	Copies the sign of *y* to *x*.
fabs (*x*)	Returns the absolute value (positive form) of *x*.

Because these functions only manipulate the sign bit of the value and do not try to compute the value at all, they raise no floating-point exceptions.

copysign

You can use the `copysign` function to assign to some real number the sign of a second value.

```
double_t copysign (double_t x, double_t y);
long double copysignl (long double x, long double y);
```

`x`	Any floating-point number.
`y`	Any floating-point number.

DESCRIPTION

The `copysign` function copies the sign of the `y` parameter into the `x` parameter and returns the resulting number.

`copysign(x, 1.0)` is always the absolute value of `x`. The `copysign` function simply manipulates sign bits and hence raises no exception flags.

EXCEPTIONS

When x and y are finite and nonzero, the result of copysign (x, y) is exact.

SPECIAL CASES

Table 10-5 shows the results when one of the arguments to the `copysign` function is a zero, a NaN, or an Infinity. In this table, x and y are finite, nonzero floating-point numbers.

Table 10-5 Special cases for the `copysign` function

Operation	Result	Exceptions raised
copysign $(+0, y)$	0 with sign of y	None
copysign $(x, +0)$	$\lvert x \rvert$	None
copysign $(-0, y)$	0 with sign of y	None
copysign $(x, -0)$	$-\lvert x \rvert$	None
copysign (NaN, y)	NaN with sign of y	None*
copysign (x, NaN)	x with sign of NaN	None*
copysign $(+\infty, y)$	∞ with sign of y	None
copysign $(x, +\infty)$	$\lvert x \rvert$	None
copysign $(-\infty, y)$	∞ with sign of y	None
copysign $(x, -\infty)$	$-\lvert x \rvert$	None

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = copysign(–1234.567, 1.0);/* z = 1234.567 */
z = copysign(1.0, –1234.567);/* z = –1.0 */
```

fabs

You can use the `fabs` function to determine the absolute value of a real number.

```
double_t fabs (double_t x);
long double fabsl (long double x);
```

`x` Any floating-point number.

DESCRIPTION

The `fabs` function returns the absolute value (positive value) of its argument.

fabs $(x) = |x|$

This function looks only at the sign bit, not the value, of its argument.

EXCEPTIONS

When x is finite and nonzero, the result of fabs (x) is exact.

SPECIAL CASES

Table 10-6 shows the results when the argument to the `fabs` function is a zero, a NaN, or an Infinity.

Table 10-6 Special cases for the `fabs` function

Operation	Result	Exceptions raised
fabs (+0)	+0	None
fabs (−0)	+0	None
fabs (NaN)	NaN	None*
fabs (+∞)	+∞	None
fabs (−∞)	+∞	None

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = fabs(–1.0);   /* z = 1 */
z = fabs(245.0);  /* z = 245 */
```

Exponential Functions

MathLib provides six exponential functions:

exp (x)	The base e or natural exponential e^x.
exp2 (x)	The base 2 exponential 2^x.
expm1 (x)	The base e exponential minus 1.
ldexp (x, n)	Returns $x \times 2^n$ (equivalent to scalb).
pow (x, y)	Returns x^y.
scalb (x, n)	Returns $x \times 2^n$.

exp

You can use the `exp` function to raise e to some power.

```
double_t exp (double_t x);
```

`x` Any floating-point number.

DESCRIPTION

The `exp` function performs the exponential function on its argument.

$$\exp(x) = e^x$$

The `log` function performs the inverse operation $(\ln e^x)$.

EXCEPTIONS

When x is finite and nonzero, the result of exp (x) might raise the following exceptions:

- inexact (for all finite, nonzero values of x)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 10-7 shows the results when the argument to the exp function is a zero, a NaN, or an Infinity.

Table 10-7 Special cases for the exp function

Operation	Result	Exceptions raised
exp (+0)	+1	None
exp (−0)	+1	None
exp (NaN)	NaN	None*
exp (+∞)	+∞	None
exp (−∞)	+0	None

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = exp(0.0);  /* z = e^0 = 1. */
z = exp(1.0);  /* z = e^1 ≈ 2.71828128...The inexact exception is
                  raised.  /
```

exp2

You can use the exp2 function to raise 2 to some power.

```
double_t exp2 (double_t x);
```

x Any floating-point number.

DESCRIPTION

The exp2 function returns the base 2 exponential of its argument.

$$\text{exp2}\,(x) = 2^x$$

The log2 function performs the inverse operation $(\log_2 2^x)$.

EXCEPTIONS

When *x* is finite and nonzero, the result of exp2 (*x*) might raise the following exceptions:

- inexact (for all finite, nonzero values of *x*)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 10-8 shows the results when the argument to the `exp2` function is a zero, a NaN, or an Infinity.

Table 10-8 Special cases for the `exp2` function

Operation	Result	Exceptions raised
exp2 (+0)	+1	None
exp2 (−0)	+1	None
exp2 (NaN)	NaN	None*
exp2 (+∞)	+∞	None
exp2 (−∞)	+0	None

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = exp2(2.0); /* z = 2^2 = 4. The inexact exception is raised. */
z = exp2(1.5); /* z = 2^1.5 ≈ 2.82843. The inexact exception is
                  raised. */
```

expm1

You can use the `expm1` function to raise *e* to some power and subtract 1.

```
double_t expm1 (double_t x);
```

`x` Any floating-point number.

DESCRIPTION

The `expm1` function returns the natural exponential decreased by 1.

$$\text{expm1}(x) = e^x - 1$$

For small numbers, use the function call `expm1(x)` instead of the expression

`exp(x) – 1`

The call `expm1(x)` produces a more exact result because it avoids the roundoff error that might occur when the expression is computed.

EXCEPTIONS

When x is finite and nonzero, the result of $\text{expm1}(x)$ might raise the following exceptions:

- inexact (for all finite, nonzero values of x)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 10-9 shows the results when the argument to the `expm1` function is a zero, a NaN, or an Infinity.

Table 10-9 Special cases for the `expm1` function

Operation	Result	Exceptions raised
expm1 (+0)	+0	None
expm1 (−0)	–0	None
expm1 (NaN)	NaN	None*
expm1 (+∞)	+∞	None
expm1 (−∞)	–1	None

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = expm1(–2.1);      /* z = e^–2.1 – 1 = –0.877544. The inexact
                         exception is raised. */
z = expm1(6);         /* z = e^6 – 1 = 402.429. The inexact
                         exception is raised. */
```

ldexp

You can use the `ldexp` function to perform efficient scaling by a power of 2.

```
double_t ldexp (double_t x, int n);
```

`x`	Any floating-point number.
`n`	An integer representing a power of 2 by which `x` should be multiplied.

DESCRIPTION

The `ldexp` function computes the value $x \times 2^n$ without computing 2^n. This is an ANSI standard C library function.

$$\text{ldexp}\ (x, n) = x \times 2^n$$

The `scalb` function (described on page 10-19) performs the same operation as this function. The `frexp` function performs the inverse operation; that is, it splits `x` into its fraction field and exponent field.

EXCEPTIONS

When x is finite and nonzero, either the result of $\text{ldexp}\ (x, n)$ is exact or it raises one of the following exceptions:

- inexact (if an overflow or underflow occurs)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 10-10 shows the results when the floating-point argument to the `ldexp` function is a zero, a NaN, or an Infinity. In this table, n is any integer.

Table 10-10 Special cases for the `ldexp` function

Operation	Result	Exceptions raised
$\text{ldexp}\ (+0, n)$	+0	None
$\text{ldexp}\ (-0, n)$	−0	None
$\text{ldexp}\ (\text{NaN}, n)$	NaN	None[*]
$\text{ldexp}\ (+\infty, n)$	+∞	None
$\text{ldexp}\ (-\infty, n)$	−∞	None

[*] If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = ldexp(3.0, 3);   /* z = 3 × 2³ = 24 */
z = ldexp(0.0, 3);   /* z = 0 × 2³ = 0 */
```

pow

You can use the pow function to raise a real number to the power of some other real number.

```
double_t pow (double_t x, double_t y);
```

x	Any floating-point number.
y	Any floating-point number.

DESCRIPTION

The pow function computes x to the y power. This is an ANSI standard C library function.

$$\text{pow}\ (x, y) = x^y$$

Use the function call pow(x,y) instead of the expression

```
exp(y * log(x))
```

The call pow(x,y) produces a more exact result.

There are some differences between this implementation and the behavior of the pow function in a SANE implementation. For example, in SANE pow(NAN,0) returns a NaN, whereas in PowerPC Numerics, pow(NAN,0) returns a 1.

EXCEPTIONS

When x and y are finite and nonzero, either the result of $\text{pow}\ (x, y)$ is exact or it raises one of the following exceptions:

- inexact (if y is not an integer or an underflow or overflow occurs)
- invalid (if x is negative and y is not an integer)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 10-11 shows the results when one of the arguments to the pow function is a zero, a NaN, or an Infinity, plus other special cases for the pow function. In this table, x and y are finite, nonzero floating-point numbers.

Table 10-11 Special cases for the pow function

Operation	Result	Exceptions raised
pow (x, y) for $x < 0$	NaN if y is not integer	Invalid
	x^y if y is integer	None
pow $(+0, y)$	±0 if y is odd integer > 0	None
	+0 if $y > 0$ but not odd integer	None
	±∞ if y is odd integer < 0	Divide-by-zero
	+∞ if $y < 0$ but not odd integer	Divide-by-zero
pow $(x, +0)$	+1	None
pow $(-0, y)$	±0 if y is odd integer > 0	None
	+0 if $y > 0$ but not odd integer	None
	±∞ if y is odd integer < 0	Divide-by-zero
	+∞ if $y < 0$ but not odd integer	Divide-by-zero
pow $(x, -0)$	+1	None
pow (NaN, y)	NaN if $y \neq 0$	None*
	+1 if $y = 0$	None*
pow (x, NaN)	NaN	None*
pow $(+\infty, y)$	+∞ if $y > 0$	None
	+0 if $y < 0$	None
	+1 if $y = 0$	None
pow $(x, +\infty)$	+∞ if $\lvert x \rvert > 1$	None
	+0 if $\lvert x \rvert < 1$	None
	NaN if $\lvert x \rvert = 1$	Invalid
pow $(-\infty, y)$	−∞ if y is odd integer > 0	None
	+∞ if $y > 0$ but not odd integer	None
	−0 if y is odd integer < 0	None
	+0 if $y < 0$ but not odd integer	None
	+1 if $y = 0$	None
pow $(x, -\infty)$	+0 if $\lvert x \rvert > 1$	None
	+∞ if $\lvert x \rvert < 1$	None
	NaN if $\lvert x \rvert = 1$	Invalid

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = pow(NAN, 0);     /* z = 1 */
```

scalb

You can use the `scalb` function to perform efficient scaling by a power of 2.

```
double_t scalb (double_t x, long int n);
```

`x`	Any floating-point number.
`n`	An integer representing a power of 2 by which `x` should be multiplied.

DESCRIPTION

The `scalb` function performs efficient scaling of its floating-point argument by a power of 2.

$$\text{scalb}\ (x, n) = x \times 2^{n}$$

Using the `scalb` function is more efficient than performing the actual arithmetic.

This function performs the same operation as the `ldexp` transcendental function described on page 10-16.

EXCEPTIONS

When x is finite and nonzero, either the result of scalb (x, n) is exact or it raises one of the following exceptions:

- inexact (if the result causes an overflow or underflow exception)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 10-12 shows the results when the floating-point argument to the `scalb` function is a zero, a NaN, or an Infinity. In this table, n is any integer.

Table 10-12 Special cases for the `scalb` function

Operation	Result	Exceptions raised
scalb $(+0, n)$	+0	None
scalb $(-0, n)$	−0	None
scalb (NaN, n)	NaN	None[*]
scalb $(+\infty, n)$	+∞	None
scalb $(-\infty, n)$	−∞	None

[*] If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = scalb(1, 3);  /* z = 1 × 2³ = 8 */
```

Logarithmic Functions

MathLib provides seven logarithmic functions:

frexp (x, exp)	Splits x into fraction and exponent fields.
log (x)	Base e or natural logarithm.
log10 (x)	Base 10 logarithm.
log1p (x)	Computes $\log(1 + x)$.
log2 (x)	Base 2 logarithm.
logb (x)	Returns exponent part of x.
modf $(x, iptr)$	Splits x into an integer and a fraction.

frexp

You can use the `frexp` function to find out the values of a floating-point number's fraction field and exponent field.

```
double_t frexp (double_t x, int *exponent);
```

`x`	Any floating-point number.
`exponent`	A pointer to an integer in which the value of the exponent can be returned.

DESCRIPTION

The `frexp` function splits its first argument into a fraction part and a base 2 exponent part. This is an ANSI standard C library function.

$$\text{frexp}\ (x, n)\ = f\ \text{ such that } x\ = f \times 2^n$$

or

$$\text{frexp}\ (x, n)\ = f\ \text{ such that } n\ =\ (1 + \text{logb}\,(x))\ \text{ and } f\ =\ \text{scalb}\ (x, -n)$$

The return value of `frexp` is the value of the fraction field of the argument `x`. The exponent field of `x` is stored in the address pointed to by the `exponent` argument.

For finite nonzero inputs, `frexp` returns either 0.0 or a value whose magnitude is between 0.5 and 1.0.

The `ldexp` and `scalb` functions perform the inverse operation (compute $f \times 2^n$).

EXCEPTIONS

If x is finite and nonzero, the result of $\text{frexp}\ (x, n)$ is exact.

SPECIAL CASES

Table 10-13 shows the results when the input argument to the `frexp` function is a zero, a NaN, or an Infinity.

Table 10-13 Special cases for the `frexp` function

Operation	Result	Exceptions raised
$\text{frexp}\ (+0, n)$	+0 (n = 0)	None
$\text{frexp}\ (-0, n)$	−0 (n = 0)	None
$\text{frexp}\ (\text{NaN}, n)$	NaN (n is undefined)	None*
$\text{frexp}\ (+\infty, n)$	+∞ (n is undefined)	None
$\text{frexp}\ (-\infty, n)$	−∞ (n is undefined)	None

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = frexp(2E300, n);    /* z ≈ 0.746611 and n = 998. In other
                           words, 2 × 10^300 ≈ 0.746611 × 2^998. */
```

log

You can use the `log` function to compute the natural logarithm of a real number.

```
double_t log (double_t x);
```

`x` Any positive floating-point number.

DESCRIPTION

The `log` function returns the natural (base *e*) logarithm of its argument.

$$\log(x) = \log_e x = \ln x = y \text{ such that } x = e^y$$

The `exp` function performs the inverse (exponential) operation.

EXCEPTIONS

When *x* is finite and nonzero, the result of $\log(x)$ might raise one of the following exceptions:

- inexact (for all finite, nonzero values of *x* other than +1)
- invalid (if *x* is negative)

SPECIAL CASES

Table 10-14 shows the results when the argument to the `log` function is a zero, a NaN, or an Infinity, plus other special cases for the `log` function.

Table 10-14 Special cases for the `log` function

Operation	Result	Exceptions raised
$\log(x)$ for $x < 0$	NaN	Invalid
$\log(+1)$	+0	None
$\log(+0)$	$-\infty$	Divide-by-zero
$\log(-0)$	$-\infty$	Divide-by-zero
log (NaN)	NaN	None[*]
$\log(+\infty)$	$+\infty$	None
$\log(-\infty)$	NaN	Invalid

[*] If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = log(+1.0);    /* z = +0.0 because e^0 = 1 */
z = log(–1.0);    /* z = NAN because negative arguments are not
                     allowed. The invalid exception is raised. */
```

log10

You can use the `log10` function to compute the common logarithm of a real number.

```
double_t log10 (double_t x);
```

`x` Any positive floating-point number.

DESCRIPTION

The `log10` function returns the common (base 10) logarithm of its argument.

$$\log 10 (x) = \log_{10} x = y \text{ such that } x = 10^y$$

EXCEPTIONS

When x is finite and nonzero, the result of $\log 10 (x)$ might raise one of the following exceptions:

- inexact (for all finite, nonzero values of x other than +1)
- invalid (when x is negative)

SPECIAL CASES

Table 10-15 shows the results when the argument to the `log10` function is a zero, a NaN, or an Infinity, plus other special cases for the `log10` function.

Table 10-15 Special cases for the `log10` function

Operation	Result	Exceptions raised
$\log 10(x)$ for $x < 0$	NaN	Invalid
$\log 10(+1)$	+0	None
$\log 10(+0)$	–∞	Divide-by-zero
$\log 10(-0)$	–∞	Divide-by-zero
$\log 10(\mathrm{NaN})$	NaN	None*
$\log 10(+\infty)$	+∞	None
$\log 10(-\infty)$	NaN	Invalid

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = log10(+1.0);  /* z = 0.0 because 10^0 = 1 */
z = log10(10.0);  /* z = 1.0 because 10^1 = 10. The inexact
                     exception is raised. */
z = log10(–1.0);  /* z = NAN because negative arguments are not
                     allowed. The invalid exception is raised. */
```

log1p

You can use the `log1p` function to compute the natural logarithm of 1 plus a real number.

```
double_t log1p (double_t x);
```

`x` Any floating-point number greater than –1.

DESCRIPTION

The `log1p` function computes the natural logarithm of 1 plus its argument.

$$\mathrm{log1p}\,(x) = \log_e\,(x+1) = \ln\,(x+1) = y \text{ such that } 1+x = 10^y$$

For small numbers, use the function call `log1p(x)` instead of the function call `log(1 + x)`. The call `log1p(x)` produces a more exact result because it avoids the roundoff error that might occur when the expression `1 + x` is computed.

EXCEPTIONS

When x is finite and nonzero, the result of log1p (x) might raise one of the following exceptions:

- inexact (for all finite, nonzero values of $x > -1$)
- invalid (when x is less than –1)
- divide-by-zero (when x is –1)

SPECIAL CASES

Table 10-16 shows the results when the argument to the `log1p` function is a zero, a NaN, or an Infinity, plus other special cases for the `log1p` function.

Table 10-16 Special cases for the `log1p` function

Operation	Result	Exceptions raised
log1p (x) for $x < -1$	NaN	Invalid
log1p (−1)	–∞	Divide-by-zero
log1p (+0)	+0	None
log1p (−0)	–0	None
log1p (NaN)	NaN	None*
log1p (+∞)	+∞	None
log1p (−∞)	NaN	Invalid

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = log1p(–1.0);  /* z = log(0) = –INFINITY. The divide-by-zero
                     and inexact exceptions are raised. */
z = log1p(0.0);   /* z = log(1) = 0.0 because e^0 = 1. */
z = log1p(–2.0);  /* z = log(–1) = NAN because logarithms of
                     negative numbers are not allowed. The
                     invalid exception is raised. */
```

log2

You can use the `log2` function to compute the binary logarithm of a real number.

```
double_t log2 (double_t x);
```

`x` Any positive floating-point number.

DESCRIPTION

The `log2` function returns the binary (base 2) logarithm of its argument.

$\log 2(x) = \log_2 x = y$ such that $x = 2^y$

The `exp2` function performs the inverse operation.

EXCEPTIONS

When *x* is finite and nonzero, the result of $\log 2(x)$ might raise one of the following exceptions:

- inexact (for all finite, nonzero values of *x* other than +1)
- invalid (when *x* is negative)

SPECIAL CASES

Table 10-17 shows the results when the argument to the `log2` function is a zero, a NaN, or an Infinity, plus other special cases for the `log2` function.

Table 10-17 Special cases for the `log2` function

Operation	Result	Exceptions raised
$\log 2(x)$ for $x < 0$	NaN	Invalid
$\log 2(+1)$	+0	None
$\log 2(+0)$	−∞	Divide-by-zero
$\log 2(-0)$	−∞	Divide-by-zero
$\log 2(\text{NaN})$	NaN	None*
$\log 2(+\infty)$	+∞	None
$\log 2(-\infty)$	NaN	Invalid

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = log2(+1.0);   /* z = +0 because 2^0 = 1 */
z = log2(2.0);    /* z = 1 because 2^1 = 2. The inexact exception
                     is raised. */
z = log2(–1.0);   /* z = NAN because negative arguments are not
                     allowed. The invalid exception is raised.*/
```

logb

You can use the `logb` function to determine the value in the exponent field of a floating-point number.

```
double_t logb (double_t x);
```

`x`	Any floating-point number.

DESCRIPTION

The `logb` function returns the signed exponent of its argument *x* as a signed integer value.

$$\mathrm{logb}\,(x) = y \text{ such that } x = f \times 2^{y}$$

When the argument is a denormalized number, the exponent is determined as if the input argument had first been normalized.

Note that for a nonzero finite *x*, $1 \le \mathrm{fabs}\,(\mathrm{scalb}\,(x, -\mathrm{logb}\,(x))) < 2$.

That is, for a nonzero finite *x*, the magnitude of *x* taken to the power of its inverse exponent is between 1 and 2.

This function conforms to IEEE Standard 854, which differs from IEEE Standard 754 on the treatment of a denormalized argument *x*.

EXCEPTIONS

If *x* is finite and nonzero, the result of $\mathrm{logb}\,(x)$ is exact.

SPECIAL CASES

Table 10-18 shows the results when the argument to the `logb` function is a zero, a NaN, or an Infinity.

Table 10-18 Special cases for the `logb` function

Operation	Result	Exceptions raised
$\mathrm{logb}(+0)$	$-\infty$	Divide-by-zero
$\mathrm{logb}(-0)$	$-\infty$	Divide-by-zero
$\mathrm{logb}(\mathrm{NaN})$	NaN	None*
$\mathrm{logb}(+\infty)$	$+\infty$	None
$\mathrm{logb}(-\infty)$	$+\infty$	None

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = logb(789.9);  /* z = 9.0 because 789.9 ≈ 1.54 × 2^9 */
z = logb(21456789);/* z = 24.0 because 21456789 ≈ 1.28 × 2^24 */
```

modf

You can use the `modf` function to split a real number into a fractional part and an integer part.

```
float modff (float x, float *iptrf);
double modf (double x, double *iptr);
```

`x`	Any floating-point number.
`iptr`	A pointer to a floating-point variable in which the integer part can be stored upon return.

DESCRIPTION

The `modf` function splits its first argument into a fractional part and an integer part. This is an ANSI standard C function.

$$\mathrm{modf}(x, n) = f \texttt{ such that } |f| < 1.0 \texttt{ and } f + n = x$$

The fractional part is returned as the value of the function, and the integer part is stored as a floating-point number in the area pointed to by `iptr`. The fractional part and the integer part both have the same sign as the argument `x`.

EXCEPTIONS

If *x* is finite and nonzero, the result of modf (*x*, *n*) is exact.

SPECIAL CASES

Table 10-19 shows the results when the floating-point argument to the `modf` function is a zero, a NaN, or an Infinity.

Table 10-19 Special cases for the `modf` function

Operation	Result	Exceptions raised
modf (+0, *n*)	+0 (*n* = 0)	None
modf (−0, *n*)	−0 (*n* = 0)	None
modf (NaN, *n*)	NaN (*n* = NaN)	None[*]
modf (+∞, *n*)	+0 (*n* = +∞)	None
modf (−∞, *n*)	−0 (*n* = −∞)	None

[*] If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = modf(1.0, n); /* z = 0.0 and n = 1.0 */
z = modf(+INFINITY, n); /* z = 0.0 and n = +INFINITY because the
                           value +∞ is an integer. */
```

Trigonometric Functions

MathLib provides the following **trigonometric functions**:

cos (*x*)	Computes the cosine of x.
sin (*x*)	Computes the sine of *x*.
tan (*x*)	Computes the tangent of *x*.
acos (*x*)	Computes the arc cosine of *x*.
asin (*x*)	Computes the arc sine of *x*.
atan (*x*)	Computes the arc tangent of *x*.
atan2 (*y*, *x*)	Computes the arc tangent of *y* / *x*.

The remaining trigonometric functions can be computed easily and efficiently from the transcendental functions provided.

The arguments for trigonometric functions (cos, sin, and tan) and return values for inverse trigonometric functions (acos, asin, atan, and atan2) are expressed in radians. The cosine, sine, and tangent functions use an argument reduction based on the remainder function (see Chapter 6, "Numeric Operations and Functions") and the constant pi, where pi is the nearest approximation of π with 53 bits of precision. The cosine, sine, and tangent functions are periodic with respect to the constant pi, so their periods are different from their mathematical counterparts and diverge from their counterparts when their arguments become very large.

cos

You can use the cos function to compute the cosine of a real number.

```
double_t cos (double_t x);
```

x Any finite floating-point number.

DESCRIPTION

The cos function returns the cosine of its argument. The argument is the measure of an angle expressed in radians. This function is **symmetric** with respect to the y-axis (cos x = cos $-x$).

The acos function performs the inverse operation $(\arccos(y))$.

EXCEPTIONS

When x is finite and nonzero, $\cos(x)$ raises the inexact exception.

SPECIAL CASES

Table 10-20 shows the results when the argument to the cos function is a zero, a NaN, or an Infinity, plus other special cases for the cos function.

Table 10-20 Special cases for the cos function

Operation	Result	Exceptions raised
cos (π)	–1	Inexact
cos (+0)	1	None
cos (−0)	1	None
cos (NaN)	NaN	None*
cos (+∞)	NaN	Invalid
cos (−∞)	NaN	Invalid

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = cos(0);    /* z = 1.0. */
z = cos(pi/2); /* z = –0.0. The inexact exception is raised. */
z = cos(pi);   /* z = –1.0. The inexact exception is raised. */
z = cos(–pi/2);/* z = 0.0. The inexact exception is raised. */
z = cos(–pi);  /* z = –1.0. The inexact exception is raised. */
```

sin

You can use the `sin` function to compute the sine of a real number.

```
double_t sin (double_t x);
```

`x`	Any finite floating-point number.

DESCRIPTION

The `sin` function returns the sine of its argument. The argument is the measure of an angle expressed in radians. This function is **antisymmetric** with respect to the y-axis (sin $x \neq$ sin $-x$).

The `asin` function performs the inverse operation $(\arcsin(y))$.

EXCEPTIONS

When x is finite and nonzero, the result of $\sin(x)$ might raise one of the following exceptions:

- inexact (for all finite, nonzero values of x)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 10-21 shows the results when the argument to the `sin` function is a zero, a NaN, or an Infinity, plus other special cases for the `sin` function.

Table 10-21 Special cases for the `sin` function

Operation	Result	Exceptions raised
sin (π)	0	Inexact
sin (+0)	+0	None
sin (−0)	–0	None
sin (NaN)	NaN	None*
sin (+∞)	NaN	Invalid
sin (−∞)	NaN	Invalid

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = sin(pi/2);    /* z = 1. The inexact exception is raised. */
z = sin(pi);      /* z = 0. The inexact exception is raised. */
z = sin(–pi/2);   /* z = –1. The inexact exception is raised. */
z = sin(–pi);     /* z = 0. The inexact exception is raised. */
```

tan

You can use the `tan` function to compute the tangent of a real number.

```
double_t tan (double_t x);
```

`x`	Any finite floating-point number.

DESCRIPTION

The `tan` function returns the tangent of its argument. The argument is the measure of an angle expressed in radians. This function is antisymmetric.

The `atan` function performs the inverse operation $(\arctan(y))$.

EXCEPTIONS

When *x* is finite and nonzero, the result of $\tan(x)$ might raise one of the following exceptions:

- inexact (for all finite, nonzero values of *x*)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 10-22 shows the results when the argument to the `tan` function is a zero, a NaN, or an Infinity, plus other special cases for the `tan` function.

Table 10-22 Special cases for the `tan` function

Operation	Result	Exceptions raised
$\tan(\pi)$	0	Inexact
$\tan(\pi/2)$	$+\infty$	Inexact
$\tan(+0)$	+0	None
$\tan(-0)$	−0	None
$\tan(\text{NaN})$	NaN	None[*]
$\tan(+\infty)$	NaN	Invalid
$\tan(-\infty)$	NaN	Invalid

[*] If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = tan(pi);   /* z = 0. The inexact exception is raised. */
z = tan(pi/2); /* z = +INFINITY. The inexact exception is
                  raised. */
z = tan(pi/4); /* z = 1. The inexact exception is raised. */
```

acos

You can use the `acos` function to compute the arc cosine of a real number between –1 and +1.

```
double_t acos (double_t x);
```

`x` Any floating-point number in the range $-1 \le x \le 1$.

DESCRIPTION

The `acos` function returns the arc cosine of its argument *x*. The return value is expressed in radians in the range [0, π].

$\operatorname{acos}(x) = \arccos(x) = y$ such that $\cos(y) = x$ for $-1 \le x \le 1$

The `cos` function performs the inverse operation $(\cos(y))$.

EXCEPTIONS

When x is finite and nonzero, the result of acos (x) might raise one of the following exceptions:

- inexact (for all finite, nonzero values of x other than 1)
- invalid (if $|x| > 1$)

SPECIAL CASES

Table 10-23 shows the results when the argument to the `acos` function is a zero, a NaN, or an Infinity, plus other special cases for the `acos` function.

Table 10-23 Special cases for the `acos` function

Operation	Result	Exceptions raised
acos (x) for $\|x\| > 1$	NaN	Invalid
acos (−1)	π	Inexact
acos (+1)	+0	None
acos (+0)	π/2	Inexact
acos (−0)	π/2	Inexact
acos (NaN)	NaN	None[*]
acos (+∞)	NaN	Invalid
acos (−∞)	NaN	Invalid

[*] If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = acos(1.0);     /* z = arccos (1) = 0.0 */
z = acos(–1.0);    /* z = arccos (–1) = π. The inexact exception is
                      raised. */
```

asin

You can use the `asin` function to compute the arc sine of a real number between –1 and 1.

```
double_t asin (double_t x);
```

`x` Any floating-point number in the range $–1 \le x \le 1$.

DESCRIPTION

The `asin` function returns the arc sine of its argument. The return value is expressed in radians in the range $[-\pi/2, +\pi/2]$. This function is antisymmetric.

asin (x) = arcsin (x) = y such that sin (y) = x for $-1 \le x \le 1$

The `sin` function performs the inverse operation $(\sin(y))$.

EXCEPTIONS

When x is finite and nonzero, the result of asin (x) might raise one of the following exceptions:

- inexact (for all finite, nonzero values of x)
- invalid (if $|x| > 1$)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 10-24 shows the results when the argument to the `asin` function is a zero, a NaN, or an Infinity, plus other special cases for the `asin` function.

Table 10-24 Special cases for the `asin` function

Operation	Result	Exceptions raised
asin (x) for $\|x\| > 1$	NaN	Invalid
asin (-1)	$-\pi/2$	Inexact
asin $(+1)$	$\pi/2$	Inexact
asin $(+0)$	+0	None
asin (-0)	−0	None
asin (NaN)	NaN	None*
asin $(+\infty)$	NaN	Invalid
asin $(-\infty)$	NaN	Invalid

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = asin(1.0);    /* z = arcsin 1 = π/2. The inexact exception is
                     raised. */
z = asin(–1.0);   /* z = arcsin –1 = –π/2. The inexact exception
                     is raised. */
```

atan

You can use the `atan` function to compute the arc tangent of a real number.

```
double_t atan (double_t x);
```

`x`	Any floating-point number.

DESCRIPTION

The `atan` function returns the arc tangent of its argument. The return value is expressed in radians in the range $[-\pi/2, +\pi/2]$. This function is antisymmetric.

$$\operatorname{atan}(x) = \arctan(x) = y \texttt{ such that } \tan(y) = x \texttt{ for all } x$$

The `tan` function performs the inverse operation $(\tan(y))$.

EXCEPTIONS

When x is finite and nonzero, the result of $\operatorname{atan}(x)$ might raise one of the following exceptions:

- inexact (for all nonzero values of x)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 10-25 shows the results when the argument to the `atan` function is a zero, a NaN, or an Infinity.

Table 10-25 Special cases for the `atan` function

Operation	Result	Exceptions raised
$\operatorname{atan}(+0)$	$+0$	None
$\operatorname{atan}(-0)$	-0	None
$\operatorname{atan}(\mathrm{NaN})$	NaN	None[*]
$\operatorname{atan}(+\infty)$	$+\pi/2$	Inexact
$\operatorname{atan}(-\infty)$	$-\pi/2$	Inexact

[*] If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = atan(1.0);     /* z = arctan 1 = π/4 */
z = atan(–1.0);    /* z = arctan –1 = –π/4. The inexact exception
                      is raised. */
```

atan2

You can use the `atan2` function to compute the arc tangent of a real number divided by another real number.

```
double_t atan2 (double_t y, double_t x);
```

`y`	Any floating-point number.
`x`	Any floating-point number.

DESCRIPTION

The `atan2` function returns the arc tangent of its first argument divided by its second argument. The return value is expressed in radians in the range [–π, +π], using the signs of its operands to determine the quadrant.

$$\mathrm{atan2}\,(y, x) = \arctan\,(y/x) = z \text{ such that } \tan\,(z) = y/x$$

EXCEPTIONS

When x and y are finite and nonzero, the result of $\mathrm{atan2}\,(y, x)$ might raise one of the following exceptions:

- inexact (if either x or y is any finite, nonzero value)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 10-26 shows the results when one of the arguments to the `atan2` function is a zero, a NaN, or an Infinity. In this table, *x* and *y* are finite, nonzero floating-point numbers.

Table 10-26 Special cases for the `atan2` function

Operation	Result	Exceptions raised
atan2 (+0, *x*)	+0 *x* > 0	None
	+π *x* < 0	None
atan2 (*y*, +0)	+π/2 *y* > 0	None
	–π/2 *y* < 0	None
atan2 (±0, +0)	±0	None
atan2 (−0, *x*)	−0 *x* > 0	Inexact
	–π *x* < 0	Inexact
atan2 (*y*, −0)	+π/2 *y* > 0	None
	–π/2 *y* < 0	None
atan2 (±0, −0)	±π	Inexact
atan2 (NaN, *x*)	NaN[*]	None[†]
atan2 (*y*, NaN)	NaN	None[†]
atan2 (+∞, *x*)	π/2	Inexact
atan2 (*y*, +∞)	±0	None
atan2 (±∞, +∞)	±3π/4	Inexact
atan2 (−∞, *x*)	–π/2	Inexact
atan2 (*y*, −∞)	±π	None
atan2 (±∞, −∞)	±3π/4	Inexact

[*] If both arguments are NaNs, it is undefined which one `atan2` returns.
[†] If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = atan2(1.0, 1.0); /* z = arctan 1/1 = arctan 1 = π/4. The
                        inexact exception is raised. */
z = atan2(3.5, 0.0); /* z = arctan 3.5/0 = arctan ∞ = π/2 */
```

Hyperbolic Functions

MathLib provides a core of hyperbolic and inverse hyperbolic functions.

cosh (x)	Hyperbolic cosine of *x*.
sinh (x)	Hyperbolic sine of *x*.
tanh (x)	Hyperbolic tangent of *x*.
acosh (x)	Inverse hyperbolic cosine of *x*.
asinh (x)	Inverse hyperbolic sine of *x*.
atanh (x)	Inverse hyperbolic tangent of *x*.

These functions are based on other transcendental functions and defer most exception generation to the core functions they use.

cosh

You can use the `cosh` function to compute the hyperbolic cosine of a real number.

```
double_t cosh (double_t x);
```

`x`	Any floating-point number.

DESCRIPTION

The `cosh` function returns the hyperbolic cosine of its argument. This function is symmetric.

The `acosh` function performs the inverse operation $(\operatorname{arccosh}(y))$.

EXCEPTIONS

When *x* is finite and nonzero, the result of cosh (x) might raise one of the following exceptions:

- inexact (for all finite, nonzero values of *x*)
- overflow (if the result is outside the range of the data type)

SPECIAL CASES

Table 10-27 shows the results when the argument to the `cosh` function is a zero, a NaN, or an Infinity.

Table 10-27 Special cases for the `cosh` function

Operation	Result	Exceptions raised
cosh (+0)	+1	None
cosh (−0)	+1	None
cosh (NaN)	NaN	None*
cosh (+∞)	+∞	None
cosh (−∞)	+∞	None

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = cosh(1.0);    /* z ≈ 1.54308. The inexact exception is
                     raised. */
z = cosh(–1.0);   /* z ≈ 1.54308. The inexact exception is
                     raised. */
```

sinh

You can use the `sinh` function to compute the hyperbolic sine of a real number.

```
double_t sinh (double_t x);
```

`x`	Any floating-point number.

DESCRIPTION

The `sinh` function returns the hyperbolic sine of its argument. This function is antisymmetric.

The `asinh` function performs the inverse operation $(\text{arcsinh}\,(y))$.

EXCEPTIONS

When *x* is finite and nonzero, the result of sinh (*x*) might raise one of the following exceptions:

- inexact (for all finite, nonzero values of *x*)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 10-28 shows the results when the argument to the `sinh` function is a zero, a NaN, or an Infinity.

Table 10-28 Special cases for the `sinh` function

Operation	Result	Exceptions raised
sinh (+0)	+0	None
sinh (−0)	−0	None
sinh (NaN)	NaN	None*
sinh (+∞)	+∞	None
sinh (−∞)	−∞	None

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
sinh(1.0);  /* z ≈ 1.175201. The inexact exception is raised. */
sinh(–1.0); /* z ≈ –1.175201. The inexact exception is raised. */
```

tanh

You can use the `tanh` function to compute the hyperbolic tangent of a real number.

```
double_t tanh (double_t x);
```

`x` Any floating-point number.

DESCRIPTION

The `tanh` function returns the hyperbolic tangent of its argument. The return value is in the range [–1, +1]. This function is antisymmetric.

The `atanh` function performs the inverse operation $(\text{arctanh}\,(y))$.

EXCEPTIONS

When *x* is finite and nonzero, the result of $\tanh\,(x)$ might raise one of the following exceptions:

- inexact (for all finite, nonzero values of *x*)

SPECIAL CASES

Table 10-29 shows the results when the argument to the `tanh` function is a zero, a NaN, or an Infinity.

Table 10-29 Special cases for the `tanh` function

Operation	Result	Exceptions raised
$\tanh\,(+0)$	+0	None
$\tanh\,(-0)$	−0	None
$\tanh\,(\text{NaN})$	NaN	None*
$\tanh\,(+\infty)$	+1	None
$\tanh\,(-\infty)$	–1	None

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = tanh(1.0);    /* z ≈ 0.761594. The inexact exception is
                     raised. */
z = tanh(–1.0);   /* z ≈ 0.761594. The inexact exception is
                     raised. */
```

acosh

You can use the `acosh` function to compute the inverse hyperbolic cosine of a real number.

```
double_t acosh (double_t x);
```

`x` Any floating-point number in the range $1 \le x \le +\infty$.

DESCRIPTION

The acosh function returns the inverse hyperbolic cosine of its argument. This function is antisymmetric.

$$\text{acosh}(x) = \text{arccosh } x = y \text{ such that } \cosh y = x$$

The cosh function performs the inverse operation $(\cosh(y))$.

EXCEPTIONS

When x is finite and nonzero, the result of $\text{acosh}(x)$ might raise one of the following exceptions:

- inexact (for all finite values of $x > 1$)
- invalid (if $x < 1$)

SPECIAL CASES

Table 10-30 shows the results when the argument to the acosh function is a zero, a NaN, or an Infinity, plus other special cases for the acosh function.

Table 10-30 Special cases for the acosh function

Operation	Result	Exceptions raised
$\text{acosh}(x)$ for $x < 1$	NaN	Invalid
acosh (1)	+0	None
acosh (+0)	NaN	Invalid
acosh (−0)	NaN	Invalid
acosh (NaN)	NaN	None*
acosh (+∞)	+∞	None
acosh (−∞)	NaN	Invalid

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = acosh(1.0);   /* z = +0 */
z = acosh(0.0);   /* z = NAN. The invalid exception is raised. */
```

asinh

You can use the `asinh` function to compute the inverse hyperbolic sine of a real number.

```
double_t asinh (double_t x);
```

`x` Any floating-point number.

DESCRIPTION

The `asinh` function returns the inverse hyperbolic sine of its argument. This function is antisymmetric.

$$\text{asinh}\,(x) = \text{arcsinh}\ x = y \ \text{ such that sinh } y = x$$

The `sinh` function performs the inverse operation $(\sinh(y))$.

EXCEPTIONS

When x is finite and nonzero, the result of $\text{asinh}\,(x)$ might raise one of the following exceptions:

- inexact (for all finite, nonzero values of x)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 10-31 shows the results when the argument to the `asinh` function is a zero, a NaN, or an Infinity.

Table 10-31 Special cases for the `asinh` function

Operation	Result	Exceptions raised
asinh (+0)	+0	None
asinh (−0)	−0	None
asinh (NaN)	NaN	None*
asinh (+∞)	+∞	None
asinh (−∞)	−∞	None

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = asinh(1.0);   /* z ≈ 0.881374. The inexact exception is
                     raised.*/
z = asinh(–1.0);  /* z ≈ 0.881374. The inexact exception is
                     raised.*/
```

atanh

You can use the `atanh` function to perform the inverse hyperbolic tangent of a real number.

```
double_t atanh (double_t x);
```

`x` Any floating-point number in the range $-1 \le x \le 1$.

DESCRIPTION

The `atanh` function returns the inverse hyperbolic tangent of its argument. This function is antisymmetric.

$$\text{atanh}\,(x) = \text{arctanh}\ x = y \ \text{ such that } \tanh\ y = x$$

The `tanh` function performs the inverse operation $(\tanh\,(y))$.

EXCEPTIONS

When x is finite and nonzero, the result of $\text{atanh}\,(x)$ might raise one of the following exceptions:

- inexact (for all finite, nonzero values of x other than +1 and –1)
- invalid (if $|x| > 1$)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 10-32 shows the results when the argument to the atanh function is a zero, a NaN, or an Infinity, plus other special cases for the atanh function.

Table 10-32 Special cases for the atanh function

Operation	Result	Exceptions raised
atanh (x) for $\|x\| > 1$	NaN	Invalid
atanh (−1)	−∞	None
atanh (+1)	+∞	None
atanh (+0)	+0	None
atanh (−0)	−0	None
atanh (NaN)	NaN	None*
atanh (+∞)	NaN	Invalid
atanh (−∞)	NaN	Invalid

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = atanh(1.0);   /* z = +INFINITY */
z = atanh(–1.0);  /* z = –INFINITY */
```

Financial Functions

MathLib provides two functions, compound and annuity, that can be used to solve various financial or time-value-of-money problems.

compound

You can use the compound function to determine the compound interest earned given an interest rate and period.

```
double_t compound (double_t rate, double_t periods);
```

rate The interest rate (any positive floating-point number).

periods The number of interest periods (any positive floating-point number). This argument might or might not be an integer.

DESCRIPTION

The compound function computes the compound interest earned.

$$\text{compound}\ (r, n) = (1 + r)^{n}$$

When rate is a small number, use the function call compound(rate,n) instead of the function call pow((1 + rate),n). The call compound(rate,n) produces a more exact result because it avoids the roundoff error that might occur when the expression 1 + rate is computed.

The compound function is directly applicable to computation of present and future values:

$$PV = FV \times (1 + r)^{-n} = \frac{FV}{\text{compound}\ (r, n)}$$

$$FV = PV \times (1 + r)^{n} = PV \times \text{compound}\ (r, n)$$

where *PV* is the amount of money borrowed and *FV* is the total amount that will be paid on the loan.

EXCEPTIONS

When *r* and *n* are finite and nonzero, the result of $\text{compound}\ (r, n)$ might raise one of the following exceptions:

- inexact (for all finite, nonzero values of $r > -1$)
- invalid (if $r < -1$)
- divide-by-zero (if *r* is –1 and $n < 0$)

SPECIAL CASES

Table 10-33 shows the results when one of the arguments to the compound function is a zero, a NaN, or an Infinity, plus other special cases for the compound function. In this table, *r* and *n* are finite, nonzero floating-point numbers.

Table 10-33 Special cases for the compound function

Operation	Result	Exceptions raised
$\text{compound}\ (r, n)$ for $r < -1$	NaN	Invalid
$\text{compound}\ (-1, n)$	0 if $n > 0$	None
	$+\infty$ if $n < 0$	Divide-by-zero
$\text{compound}\ (+0, n)$	1	None
$\text{compound}\ (r, +0)$	1	None

continued

Table 10-33 Special cases for the `compound` function (continued)

Operation	Result	Exceptions raised
compound (−0, *n*)	1	None
compound (*r*, −0)	1	None
compound (±0, ±∞)	NaN	Invalid
compound (NaN, *n*)	NaN[*]	None[†]
compound (*r*, NaN)	NaN	None[†]
compound (+∞, *n*)	+∞ if $n > 0$	None
	0 if $n < 0$	None
compound (*r*, +∞)	+∞	None
compound (−∞, *n*)	NaN	Invalid
compound (*r*, −∞)	0	None

[*] If both arguments are NaNs, the first NaN is returned.
[†] If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = compound(–2, 12);   /* z = NAN because a negative interest
                           rate does not make sense. The invalid
                           exception is raised.             */
z = compound(–1, –1);   /* z = +INFINITY because a negative
                           interest rate and negative loan period
                           do not make sense. The divide-by-zero
                           exception is raised.             */
z = compound(0, INFINITY);/* z = NAN. The invalid exception is
                           raised.                          */
```

annuity

You can use the `annuity` function to compute the present and future value of annuities.

```
double_t annuity (double_t rate, double_t periods);
```

`rate`	The interest rate (any positive floating-point number).
`periods`	The number of interest periods (any positive floating-point number). This argument might or might not be an integer.

DESCRIPTION

The `annuity` function computes the present and future values of annuities.

$$\text{annuity}\ (r, n)\ =\ \frac{1-(1+r)^{-n}}{r}$$

When `rate` is a small number, use the function call `annuity(rate,n)` instead of the expression:

```
(1 – compound(rate, –n)) / rate
```

The call `annuity(rate,n)` produces a more exact result because it avoids the roundoff errors that might occur when this expression is computed.

This function is directly applicable to the computation of present and future values of ordinary annuities:

$$PV\ =\ PMT \times \frac{1-(1+r)^{-n}}{r}\ =\ PMT \times \text{annuity}\ (r, n)$$

$$\begin{aligned} FV &= PMT \times \frac{1-(1+r)^{n}}{r} = PMT \times\ (1+r)\ n \times \frac{1-(1+r)^{-n}}{r} \\ &= PMT \times \text{compound}\ (r, n)\ \times \text{annuity}\ (r, n) \end{aligned}$$

where PV is the amount of money borrowed, FV is the total amount that will be paid on the loan, and PMT is the amount of one periodic payment.

EXCEPTIONS

When r and n are finite and nonzero, the result of annuity(r, n) might raise one of the following exceptions:

- inexact (for all finite, nonzero values of $r > -1$)
- invalid (if $r < -1$)
- divide-by-zero (if $r = -1$ and $n > 0$)

SPECIAL CASES

Table 10-34 shows the results when one of the arguments to the `annuity` function is a zero, a NaN, or an Infinity, plus other special cases for the `annuity` function. In this table, *r* and *n* are finite, nonzero floating-point numbers.

Table 10-34 Special cases for the `annuity` function

Operation	Result	Exceptions raised
annuity (r, n) for $r < -1$	NaN	Invalid
annuity (-1, n)	$+\infty$ if $n > 0$	Divide-by-zero
	-1 if $n < 0$	None
annuity ($+0$, n)	n	None
annuity (r, $+0$)	+0	None
annuity (-0, n)	n	None
annuity (r, -0)	+0	None
annuity (NaN, n)	NaN*	None†
annuity (r, NaN)	NaN	None†
annuity ($+\infty$, n)	0 if $n > 0$	None
	$-\infty$ if $n < 0$	None
annuity (r, $+\infty$)	$1/r$	None
annuity ($-\infty$, n)	NaN	Invalid
annuity (r, $-\infty$)	$-\infty$	None

* If both arguments are NaNs, the first NaN is returned.
† If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = annuity(–1, 5);  /* z = +INFINITY. The divide-by-zero
                        exception is raised.             */
z = annuity(–2, –2); /* z = NAN. The invalid exception
                        is raised.                       */
```

Error and Gamma Functions

MathLib provides four error and gamma functions:

erf (*x*)	Error function
erfc (*x*)	Complementary error function
gamma (*x*)	Computes Γ(*x*)
lgamma (*x*)	Computes the natural logarithm of the absolute value of gamma(*x*)

erf

You can use the `erf` function to perform the error function.

```
double_t erf (double_t x);
```

`x` Any floating-point number.

DESCRIPTION

The `erf` function computes the error function of its argument. This function is antisymmetric.

$$\mathrm{erf}\ (x) = \frac{2}{\pi}\int_{0}^{x} e^{(-t)^2}\,dt$$

EXCEPTIONS

When *x* is finite and nonzero, either the result of erf(*x*) is exact or it raises one of the following exceptions:

- inexact (if the result must be rounded or an underflow occurs)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 10-35 shows the results when the argument to the `erf` function is a zero, a NaN, or an Infinity.

Table 10-35 Special cases for the `erf` function

Operation	Result	Exceptions raised
erf (+0)	+0	None
erf (−0)	−0	None
erf (NaN)	NaN	None*
erf (+∞)	+1	None
erf (−∞)	–1	None

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = erf(1.0);      /* z ≈ 0.842701. The inexact exception is
                      raised. */
z = erf(–1.0);     /* z ≈ –0.842701. The inexact exception is
                      raised. */
```

erfc

You can use the `erfc` function to perform the complementary error function.

```
double_t erfc (double_t x);
```

`x` Any floating-point number.

DESCRIPTION

The `erfc` function computes the complementary error of its argument. This function is antisymmetric.

$$\operatorname{erfc}(x) = 1.0 - \operatorname{erf}(x)$$

For large positive numbers (around 10), use the function call `erfc(x)` instead of the expression `1.0 – erf(x)`. The call `erfc(x)` produces a more exact result.

EXCEPTIONS

When x is finite and nonzero, either the result of erfc (x) is exact or it raises one of the following exceptions:

- inexact (if the result must be rounded or an underflow occurs)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 10-36 shows the results when the argument to the `erfc` function is a zero, a NaN, or an Infinity.

Table 10-36 Special cases for the `erfc` function

Operation	Result	Exceptions raised
erfc $(+0)$	+1	None
erfc (-0)	+1	None
erfc (NaN)	NaN	None[*]
erfc $(+\infty)$	+0	None
erfc $(-\infty)$	+2	None

[*] If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = erfc(-INFINITY); /* z = 1 – erf(-∞) = 1 – -1 = +2.0 */
z = erfc(0.0);       /* z = 1 – erf(0) = 1 – 0 = 1.0 */
```

gamma

You can use the `gamma` function to perform $\Gamma(x)$.

```
double_t gamma (double_t x);
```

`x`	Any positive floating-point number.

DESCRIPTION

The gamma function performs $\Gamma(x)$.

$$\text{gamma}(x) = \Gamma(x) = \int_0^\infty e^{-t} t^{x-1} dt$$

The gamma function reaches overflow very fast as x approaches +∞. For large values, use the lgamma function instead.

EXCEPTIONS

When x is finite and nonzero, either the result of $\text{gamma}(x)$ is exact or it raises one of the following exceptions:

- inexact (if the result must be rounded or an overflow occurs)
- invalid (if x is a negative integer)
- overflow (if the result is outside the range of the data type)

SPECIAL CASES

Table 10-37 shows the results when the argument to the gamma function is a zero, a NaN, or an Infinity, plus other special cases for the gamma function.

Table 10-37 Special cases for the gamma function

Operation	Result	Exceptions raised
$\text{gamma}(x)$ for negative integer x	NaN	Invalid
gamma (+0)	NaN	Invalid
gamma (−0)	NaN	Invalid
gamma (NaN)	NaN	None*
gamma (+∞)	+∞	Overflow
gamma (−∞)	NaN	Invalid

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = gamma(-1.0);  /* z = NAN. The invalid exception is raised. */
z = gamma(6);     /* z = 120 */
```

lgamma

You can use the `lgamma` function to compute the natural logarithm of the absolute value of $\Gamma(x)$.

```
double_t lgamma (double_t x);
```

`x` Any positive floating-point number.

DESCRIPTION

The `lgamma` function computes the natural logarithm of the absolute value of $\Gamma(x)$.

$$\text{lgamma}(x) = \log_e(|\Gamma(x)|) = \ln(|\Gamma(x)|)$$

EXCEPTIONS

When x is finite and nonzero, either the result of lgamma (x) is exact or it raises one of the following exceptions:

- inexact (if the result must be rounded or an overflow occurs)
- overflow (if the result is outside the range of the data type)
- invalid (if $x \leq 0$)

SPECIAL CASES

Table 10-38 shows the results when the argument to the `lgamma` function is a zero, a NaN, or an Infinity, plus other special cases for the `lgamma` function.

Table 10-38 Special cases for the `lgamma` function

Operation	Result	Exceptions raised
lgamma (x) for $x < 0$	NaN	Invalid
lgamma (+0)	NaN	Invalid
lgamma (−0)	NaN	Invalid
lgamma (NaN)	NaN	None*
lgamma (+∞)	+∞	Overflow
lgamma (−∞)	NaN	Invalid

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = lgamma(–1.0);    /* z = NAN. The invalid exception is
                        raised. */
z = lgamma(3.41);    /* z = 1.10304. The inexact exception is
                        raised. */
```

Miscellaneous Functions

There are three remaining MathLib transcendental functions:

nextafter (*x*, *y*)	Returns next representable number after *x* in direction of *y*.
hypot (*x*)	Computes hypotenuse of a right triangle.
randomx (*x*)	A pseudorandom number generator.

nextafter

You can use the **nextafter functions** to find out the next value that can be represented after a given value in a particular floating-point type.

```
float       nextafterf (float x, float y);
double      nextafterd (double x, double y);
```

`x`	Any floating-point number.
`y`	Any floating-point number.

DESCRIPTION

The nextafter functions (one for each data type) generate the next representable neighbor of `x` in the direction of `y` in the proper format.

The floating-point values representable in single and double formats constitute a finite set of real numbers. The nextafter functions illustrate this fact by returning the next representable value.

If $x = y$, nextafter (x, y) returns *x* if *x* and *y* are not signed zeros.

EXCEPTIONS

When x and y are finite and nonzero, either the result of nextafter (x, y) is exact or it raises one of the following exceptions:

- inexact (if an overflow or underflow exception occurs)
- overflow (if x is finite and the result is infinite)
- underflow (if the result is inexact, must be represented as a denormalized number or 0, and $x \neq y$)

SPECIAL CASES

Table 10-39 shows the results when one of the arguments to a nextafter function is a zero, a NaN, or an Infinity. In this table, x and y are finite, nonzero floating-point numbers.

Table 10-39 Special cases for the nextafter functions

Operation	Result	Exceptions raised
nextafter $(+0, y)$	Next representable number in direction of y	Underflow
nextafter $(x, +0)$	Next representable number in direction of 0	None
nextafter $(-0, y)$	Next representable number in direction of y	Underflow
nextafter $(-0, +0)$	+0	None
nextafter $(x, -0)$	Next representable number in direction of 0	None
nextafter $(+0, -0)$	−0	None
nextafter (NaN, y)	NaN*	None†
nextafter (x, NaN)	NaN	None†
nextafter $(+\infty, y)$	Largest respresentable number	None
nextafter $(x, +\infty)$	Next representable number greater than x	None
nextafter $(-\infty, y)$	Smallest representable number	None
nextafter $(x, -\infty)$	Next representable number smaller than x	None

* If both arguments are NaNs, the value of the first NaN is returned.
† If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = nextafterf(1.0, +∞);/* z = 1.00000000000000000000001_2
                             ≈ 1.000000119209289551         */
z = nextafterd(1.0, +∞);/* z = 1.00000000  000000000000000001_2
                             ≈ 1.000000000000000222         */
```

hypot

You can use the `hypot` function to compute the length of a hypotenuse of a right triangle.

```
double_t hypot(double_t x, double_t y);
```

`x` Any floating-point number.

`y` Any floating-point number.

DESCRIPTION

The `hypot` function computes the square root of the sum of the squares of its arguments. This is an ANSI standard C library function.

$$\text{hypot}\ (x, y) = \sqrt{x^2 + y^2}$$

The function `hypot` performs it computation without undeserved overflow or underflow. For example, if $x^2 + y^2$ is greater than the maximum representable value of the data type but their square root is not, then no overflow occurs.

EXCEPTIONS

When x and y are finite and nonzero, either the result of hypot (x, y) is exact or it raises one of the following exceptions:

- inexact (if the result must be rounded or an overflow or underflow occurs)
- overflow (if the result is outside the range of the data type)
- underflow (if the result is inexact and must be represented as a denormalized number or 0)

SPECIAL CASES

Table 10-40 shows the results when one of the arguments to the `hypot` function is a zero, a NaN, or an Infinity. In this table, *x* and *y* are finite, nonzero floating-point numbers.

Table 10-40 Special cases for the `hypot` function

Operation	Result	Exceptions raised
hypot (+0, *y*)	|*y*|	None
hypot (*x*, +0)	|*x*|	None
hypot (−0, *y*)	|*y*|	None
hypot (*x*, −0)	|*x*|	None
hypot (NaN, *y*)	NaN	None*
hypot (*x*, NaN)	NaN	None*
hypot (NaN, ±∞)	∞	None
hypot (±∞, NaN)	∞	None
hypot (+∞, *y*)	+∞	None
hypot (*x*, +∞)	+∞	None
hypot (−∞, *y*)	+∞	None
hypot (*x*, −∞)	+∞	None

* If the NaN is a signaling NaN, the invalid exception is raised.

EXAMPLES

```
z = hypot(2.0, 2.0); /* z = sqrt(8.0) ≈ 2.82843. The inexact
                        exception is raised. */
```

randomx

You can use the `randomx` function to generate a random number.

```
double_t randomx (double_t * x);
```

`x` The address of an integer in the range $1 \le x \le 2^{31} - 2$ stored as a floating-point number.

DESCRIPTION

The `randomx` function is a pseudorandom number generator. The function `randomx` returns a pseudorandom number in the range of its argument. It uses the iteration formula

$$x \leftarrow (75 \times x) \bmod (2^{31} - 1)$$

If seed values of `x` are not integers or are outside the range specified for `x`, then results are unspecified. A pseudorandom rectangular distribution on the interval (0, 1) can be obtained by dividing the results from `randomx` by

$$2^{31} - 1 = \mathrm{scalb}\ (31, 1) - 1$$

EXCEPTIONS

The results are unspecified if the value of x is a noninteger or is outside of the range $1 \le x \le 2^{31} - 2$

SPECIAL CASES

If x is a zero, NaN, or Infinity, the results are unspecified.

EXAMPLES

randomx (1) = any value in the range $1 \le x \le 2^{31} - 2$.

Transcendental Functions Summary

This section summarizes the transcendental functions declared in the MathLib header file `fp.h` and the constants and data types that they use.

C Summary

Constants

```
extern const double_t pi;
```

Data Types

```
typedef short relop;

enum
{
   GREATERTHAN = ((relop) (0)),
   LESSTHAN,
   EQUALTO,
   UNORDERED
};
```

Transcendental Functions

Comparison Functions

```
double_t fdim               (double_t x, double_t y);
double_t fmax               (double_t x, double_t y);
double_t fmin               (double_t x, double_t y);
relop relation              (double_t x, double_t y);
```

Sign Manipulation Functions

```
double_t copysign           (double_t x, double_t y);
double_t fabs               (double_t x);
long double copysignl       (long double x, long double y);
long double fabsl           (long double x);
```

Exponential Functions

```
double_t exp                (double_t x);
double_t exp2               (double_t x);
double_t expm1              (double_t x);
double_t ldexp              (double_t x, int n);
double_t pow                (double_t x, double_t y);
double_t scalb              (double_t x, long int n);
```

Logarithmic Functions

```
double_t frexp              (double_t x, int *exponent);
double_t log                (double_t x);
double_t log10              (double_t x);
double_t log1p              (double_t x);
double_t log2               (double_t x);
double_t logb               (double_t x);
float modff                 (float x, float *iptrf);
double modf                 (double x, double *iptr);
```

Trigonometric Functions

```
double_t cos                (double_t x);
double_t sin                (double_t x);
double_t tan                (double_t x);
double_t acos               (double_t x);
double_t asin               (double_t x);
double_t atan               (double_t x);
double_t atan2              (double_t y, double_t x);
```

Hyperbolic Functions

```
double_t cosh               (double_t x);
double_t sinh               (double_t x);
double_t tanh               (double_t x);
double_t acosh              (double_t x);
double_t asinh              (double_t x);
double_t atanh              (double_t x);
```

Financial Functions

```
double_t compound           (double_t rate, double_t periods);
double_t annuity            (double_t rate, double_t periods);
```

Error and Gamma Functions

```
double_t erf                (double_t x);
double_t erfc               (double_t x);
double_t gamma              (double_t x);
double_t lgamma             (double_t x);
```

Nextafter Functions

```
float nextafterf            (float x, float y);
double nextafterd           (double x, double y);
```

Hypotenuse Function

```
double_t hypot              (double_t x, double_t y);
```

Random Number Generator Function

```
double_t randomx            (double_t * x);
```

PART THREE

Numerics in PowerPC Assembly Language

This part summarizes the numeric features available to PowerPC assembly-language programmers. The first chapter in Part 3 describes the basics of PowerPC floating-point architecture. The rest of the chapters describe how to access numeric features in assembly language.

The PowerPC architecture contains a floating-point processor that conforms to the IEEE standard. It directly supports a subset of the floating-point data formats and the arithmetic operations described in Part 1. Numeric operations are supported through assembly-language instructions.

By reading Part 3, you should gain an understanding of how the PowerPC architecture complies with the IEEE standard. Part 3 does not teach you how to write a numeric application in assembly language; it merely summarizes the numeric features available. Refer to the Motorola *PowerPC 601 RISC Microprocessor User's Manual* for complete details on the information presented here.

If your application is written in a high-level language, you might find this part of the book useful when debugging in low-level mode. You also might find Appendix F, "PowerPC Assembly-Language Numerics Reference," useful for this purpose.

CHAPTER 11

Introduction to Assembly-Language Numerics

Contents

This chapter introduces the numeric implementation in PowerPC assembly language. It describes the basics of the floating-point architecture, showing what floating-point data formats and registers are available, what numeric operations are available in assembly language, and what load and store instructions you must use before you can perform assembly-language numeric operations. An example application using assembly-language numeric operations is shown at the end of this chapter.

Read this chapter to learn how to use the numeric assembly-language instructions described in Chapters 12 through 14.

PowerPC Floating-Point Architecture

This section describes those pieces of the PowerPC architecture used in floating-point operations, which include

- floating-point data formats
- floating-point registers
- floating-point special-purpose registers
- the Machine State Register

Floating-Point Data Formats

The PowerPC architecture supports only the single and double floating-point data formats. These formats can represent normalized numbers, denormalized numbers, zeros, NaNs, and Infinities, and are interpreted exactly as described in Chapter 2, “Floating-Point Data Formats.” The double-double data format is implemented in software and therefore is not a valid format in PowerPC hardware.

The PowerPC hardware is double-based. This means that when you load a single-format number into a register, it is automatically converted to double format. In addition, all arithmetic operations are performed on double-format numbers unless they are specifically forced to be performed on single-format numbers.

Floating-Point Registers

The PowerPC architecture contains thirty-two 64-bit floating-point registers labeled F0 through F31 (or FP0 through FP31). Because the registers are 64 bits long, they store values using the double data format.

Floating-Point Special-Purpose Registers

The two special-purpose registers that affect floating-point operations are the Floating-Point Status and Control Register and the Condition Register.

The **Floating-Point Status and Control Register (FPSCR)** is a 32-bit register that stores the current state of the floating-point environment. It specifies the current rounding direction and notes whether any floating-point exceptions are enabled and whether any floating-point exceptions have occurred.

The **Condition Register** is a 32-bit register that stores the current state of the entire PowerPC processor. It is grouped into eight 4-bit fields labeled CR0 through CR7. Field CR1 reflects the results of floating-point operations. You may also specify one of the Condition Register fields as a place to store the result of a floating-point comparison operation or the result of a floating-point environment manipulation operation.

The FPSCR and the Condition Register are discussed more fully in Chapter 12, "Assembly-Language Environmental Controls."

The Machine State Register

The **Machine State Register** is a 32-bit supervisor-level register that reflects the current state of the entire PowerPC processor. It differs from the Condition Register in that it is accessible only by supervisor-level software and in that it stores the processor state in a different way. The Machine State Register contains 3 bits that control floating-point computations:

- Bit 18 specifies whether the floating-point instructions are available. If bit 18 is 0, the processor cannot execute floating-point instructions.
- Bits 20 and 23 specify whether floating-point exceptions are enabled. If both of these bits are 0, floating-point instructions will not raise any floating-point exceptions. If either of these bits is set, instructions can raise floating-point exceptions.

Floating-Point Instructions

Most floating-point operations are performed by the PowerPC floating-point processor. Floating-point arithmetic, conversion, comparison, and other operations are supported through assembler instructions. The only basic arithmetic operations supported are add, subtract, multiply, divide, and round-to-integer. In addition to instructions that perform the basic numeric operations, PowerPC assembly language provides instructions that can perform both a multiply and an add or subtract with at most a single roundoff error (called multiply-add instructions) and instructions that manipulate the sign bit of a number. All PowerPC floating-point assembler instructions conform to the IEEE standard.

All floating-point instructions (other than load instructions) operate on data located in the floating-point registers. The data must be loaded into a floating-point registers before any operation can be performed.

Even though the floating-point registers are double format, the data can be in either single or double format. The instruction mnemonic specifies whether the data in the floating-point register is interpreted as single or double format. For example, `fadd` means add two double-format numbers, and `fadds` means add two single-format numbers.

Load and Store Instructions

Before you perform any floating-point computation, you must load a value into a floating-point register. To do this, use one of the load instructions. Load instructions load either single or double floating-point numbers from memory into floating-point registers. Store instructions take the contents of a floating-point register and store them in memory.

Load and store instructions take one of two forms depending on which address mode is used. The first form is

instr *FPR*, *D*(*GPR*)

instr Specifies which type of load or store is to be performed.

FPR A floating-point register, which is either the source or the destination for the operation, depending on whether it is a load or a store.

D A 16-bit signed integer value.

GPR A general-purpose register or the value 0.

The *D*(*GPR*) part of the instruction determines the memory address involved. If *GPR* is not 0, it is interpreted as a general-purpose register and the contents of register *GPR* are added to the value *D* to produce the memory address. If *GPR* is 0, it is interpreted as the value 0 rather than as register GPR0, so 0 is added to *D* to produce the memory address.

Load instructions of this form are interpreted as *FPR* ← (*D* + (*GPR*)), which means that the instruction loads into *FPR* the contents of the memory address obtained by adding *D* to the contents of *GPR* (unless *GPR* is 0).

Store instructions of this form are interpreted as *D* + (*GPR*) ← (*FPR*), which means that the instruction stores the contents of *FPR* at the memory address obtained by adding *D* to the contents of *GPR* (unless *GPR* is 0).

The second form for load and store operations uses a different address mode:

instr *FPR*, *GPR1*, *GPR2*

instr Specifies which type of load or store is to be performed.

FPR A floating-point register, which is either the source or the destination for the operation, depending on whether it is a load or a store.

GPR1 A general purpose register or the value 0.

GPR2 A general-purpose register.

GPR1 and *GPR2* determine the memory address involved. If *GPR1* is not 0, it is interpreted as a general-purpose register, and the contents of register *GPR1* are added to the contents of register *GPR2* to produce the memory address. If *GPR1* is 0, it is interpreted as the value 0 rather than as register GPR0, so 0 is added to the contents of register *GPR2* to produce the memory address.

Load instructions of this form are interpreted as *FPR* ← ((*GPR1*) + (*GPR2*)) unless *GPR1* is 0.

Store instructions of this form are interpreted as (*GPR1*) + (*GPR2*) ← (*FPR*) unless *GPR1* is 0.

Table 11-1 lists and describes the PowerPC load and store instructions. There are two load and two store instructions for each address mode. One version simply performs the load or store, and the other version puts the effective memory address into the general-purpose register specified in the instruction (shown as R*n* in the table).

Each of the load and store instructions has a single and a double form, making a total of eight load and eight store instructions. If the single form of a load instruction is used, the number is converted to double format before the load is performed. If the single form of a store instruction is used, the number is converted to single format before it is stored. See Chapter 13, "Assembly-Language Numeric Conversions," for more information about conversions performed during load and store operations.

None of the load and store instructions raise floating-point exceptions or make special cases of zeros, NaNs, or Infinities.

Table 11-1 Load and store floating-point instructions

Address mode	Instruction syntax	Operation
d(R*n*)	`lfd` *DST*, *n*(*GPR*)	Load double format
	`stfd` *SRC*,*n*(*GPR*)	Store double format
	`lfs` *DST*, *n*(*GPR*)	Load single format
	`stfs` *SRC*,*n*(*GPR*)	Store single format
	`lfdu` *DST*, *n*(*GPR*)	Load double format and update
	`stfdu` *SRC*,*n*(*GPR*)	Store double format and update
	`lfsu` *DST*, *n*(*GPR*)	Load single format and update
	`stfsu` *SRC*,*n*(*GPR*)	Store single format and update

Table 11-1 Load and store floating-point instructions (continued)

Address mode	Instruction syntax	Operation
R*n*,R*m*	`lfdx` *DST*, *GPR1*,*GPR2*	Load double format indexed
	`stfdx` *SRC*,*GPR1*,*GPR2*	Store double format indexed
	`lfsx` *DST*, *GPR1*,*GPR2*	Load single format indexed
	`stfsx` *SRC*,*GPR1*,*GPR2*	Store single format indexed
	`lfdux` *DST*, *GPR1*,*GPR2*	Load double format and update indexed
	`stfdux` *SRC*,*GPR1*,*GPR2*	Store double format and update indexed
	`lfsux` *DST*, *GPR1*,*GPR2*	Load single format and update indexed
	`stfsux` *SRC*,*GPR1*,*GPR2*	Store single format and update indexed

Numerics Example Using PowerPC Assembly Language

Listing 11-1 is a code example that shows when the PowerPC assembly-language numeric features might be useful. The instructions used in this example are described in the Motorola *PowerPC 601 RISC Microprocessor User's Manual.* This example evaluates the polynomial

$$x^3 + 2x^2 - 5$$

It illustrates the evaluation of a polynomial

$$c_0 x^n + c_1 x^{n-1} + \ldots + c_n$$

using Horner's recurrence

$$r \leftarrow c_0$$

$$r \leftarrow (r \times x) + c_j \quad \text{for } j = 1 \text{ to } n$$

On entry, general-purpose register GPR0 contains the degree n (<256) of the polynomial, and floating-point register F1 points to a function argument x. The coefficient table consists of $n + 1$ double-format coefficients, starting with c_0. In this particular polynomial, $n = 3$, $c_0 = 1$, $c_1 = 2$, $c_2 = 0$, and $c_3 = -5$.

Listing 11-1 Polynomial evaluation

```
r0:   equ   0                 # general-purpose register 0
r5:   equ   5                 # general-purpose register 5
f0:   equ   0                 # floating-point register 0
f1:   equ   1                 # floating-point register 1
f2:   equ   2                 # floating-point register 2
CTR:  equ   9                 # Count Register for loops

extern   polyeval{DS}         # export the routine descriptor
extern   .polyeval            # export the entry point
# put the code in a program control section
csect    polyeval{PR}

#high-level languages prepend a period to function names
.polyeval:
   lwz      r0,0(r5)          # r0 = degree
   lfd      f0,4(r5)          # f0 = leading coefficient, c0
   addic    r5,r5,4           # r5 = address of leading coeff. &c0
   mtspr    CTR,r0            # CTR = r0
loop:
   lfdu     f2,8(r5)          # f2 = next coefficient
                              # update r5 = r5 + 8
   fmadd    f0,f0,f1,f2       # f0 = f0 * f1 + f2; ...
                              # res = res * x + c[j]
   bdnz     loop              # CTR = CTR - 1, branch if CTR ≠ 0
   fmr      f1,f0             # f1 = f0
   blr                        # return through the Link Register
   nop


#
# Set up the table of contents.  It must include at least the
# exported routines. It may also contain global data or pointers
# to data.
#
polyeval_TOC:  tc    polyeval{tc}, polyeval{PR}
```

```
#
# Build a transition vector for all exported routines so they can
# be accessed through an inter-TOC call.
#
 csect   polyeval{DS}      # it’s in a separate control section
 dc.l    .polyeval         # contains the entry point
 dc.l    0                 # loader will fill in correct TOC
                           # pointer
 dc.l    0                 # save space for environment pointer
```

CHAPTER 12

Assembly-Language Environmental Controls

Contents

This chapter describes how to use assembly-language instructions to control the floating-point environment (rounding direction and exception flags) described in Chapter 4, “Environmental Controls.” The current state of the floating-point environment is stored in the Floating-Point Status and Control Register and summarized in the Condition Register. This chapter describes exactly how these two registers store the environment. Then it describes the PowerPC assembler instructions you can use to test or change the environment.

Read this chapter to learn how to access and manipulate the floating-point environment in assembly language or to learn how the PowerPC architecture stores the floating-point environment.

The Floating-Point Environment

The two special-purpose registers that reflect and control the floating-point environment are the Floating-Point Status and Control Register and the Condition Register.

The Floating-Point Status and Control Register

The Floating-Point Status and Control Register (FPSCR) is a 32-bit register that stores the current state of the floating-point environment. It specifies the current rounding direction, whether any floating-point exceptions are enabled, and whether any floating-point exceptions have occurred. Many instructions that manipulate the FPSCR operate on 4-bit fields numbered 0 through 7. Figure 12-1 highlights some of the more useful fields in the FPSCR, and Table 12-1 shows their bit assignments. For more information on floating-point instructions, see the Motorola *PowerPC 601 RISC Microprocessor User’s Manual.*

Figure 12-1 Floating-Point Status and Control Register (FPSCR)

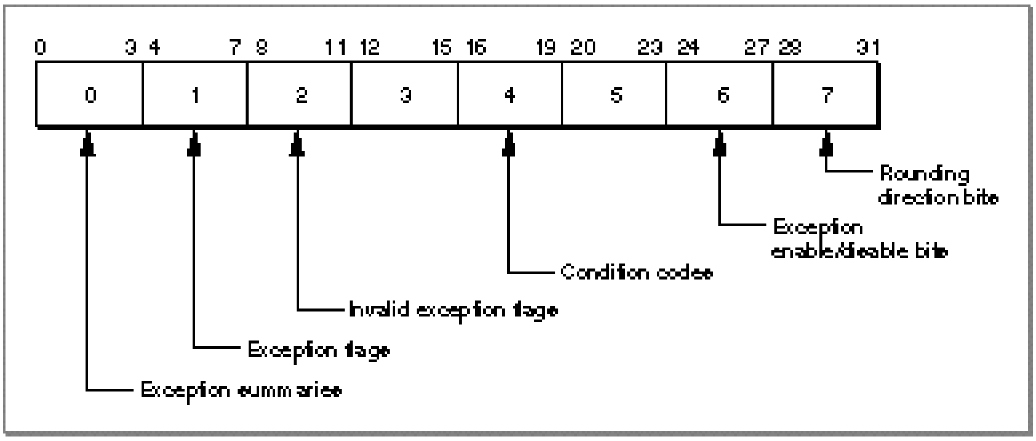


Table 12-1 Bit assignments for FPSCR fields

FPSCR field	Bit	Meaning if set
0	0	One or more of the floating-point exceptions occurred.
	1	One or more of the floating-point exceptions is enabled.
	2	One or more of the invalid exceptions occurred.
	3	An overflow exception occurred.
1	4	An underflow exception occurred.
	5	A divide-by-zero exception occurred.
	6	An inexact exception occurred.
	7	An invalid exception occurred because an operation other than load, store, move, select, or `mtfsf` was attempted on a signaling NaN.
2	8	An invalid exception occurred because $\infty - \infty$ was attempted.
	9	An invalid exception occurred because ∞ / ∞ was attempted.
	10	An invalid exception occurred because $0/0$ was attempted.
	11	An invalid exception occurred because $0 \times \infty$ was attempted.
3	12	An invalid comparison operation was attempted.
	13	The fraction field of the result has been rounded.
	14	The fraction field of the result is inexact.
	15	Class descriptor. See "Inquiries: Class and Sign" on page 12-7.
4	16	Less than or less than 0. See "Inquiries: Class and Sign" on page 12-7.
	17	Greater than or greater than 0. See "Inquiries: Class and Sign" on page 12-7.
	18	Equal to or equal to 0. See "Inquiries: Class and Sign" on page 12-7.
	19	Unordered or NaN. See "Inquiries: Class and Sign" on page 12-7.
5	20	Reserved.
	21	An invalid exception occurred because of a software request. Not implemented in MPC601.
	22	An invalid square-root operation was attempted. Not implemented in MPC601.
	23	An invalid exception occurred because of an invalid convert-to-integer operation.
6	24	The invalid exceptions are enabled.
	25	The overflow exception is enabled.
	26	The underflow exception is enabled.
	27	The divide-by-zero exception is enabled.

Table 12-1 Bit assignments for FPSCR fields (continued)

FPSCR field	Bit	Meaning if set
7	28	The inexact exception is enabled.
	29	Reserved.
	30	Rounding direction. See “Setting the Rounding Direction” on page 12-9.
	31	Rounding direction. See “Setting the Rounding Direction” on page 12-9.

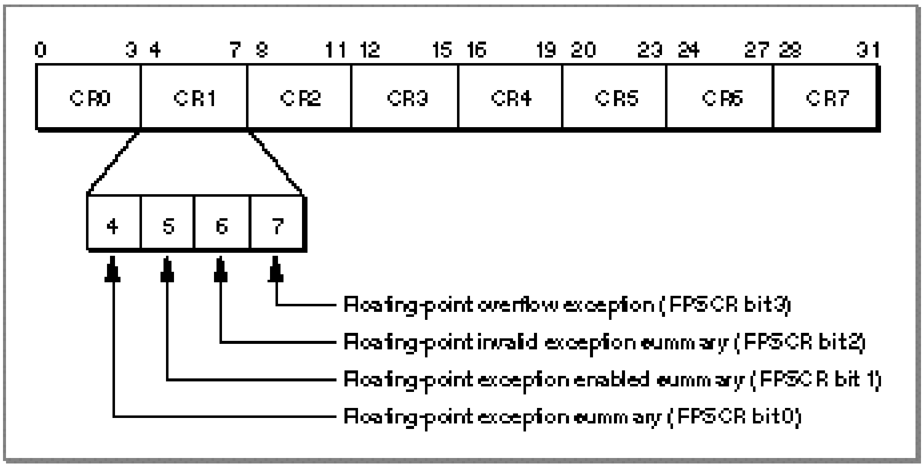
IMPORTANT

Bit 20 or 23 of the Machine State Register must be set for the FPSCR exception enable bits to be valid. For more information, see the Motorola *PowerPC 601 RISC Microprocessor User’s Manual.* s

The Condition Register

The Condition Register is a 32-bit register that stores the current state of the entire PowerPC processor. It is grouped into eight 4-bit fields labeled CR0 through CR7 (see Figure 12-2). Field CR1 (bits 4 through 7) reflects the results of floating-point operations.

Figure 12-2 Condition Register


0
3
4
7
8
11
12
15
16
19
20
23
24
27
28
31
CR0
CR1
CR2
CR3
CR4
CR5
CR6
CR7
4
5
6
7
Floating-point overflow exception (FPSCR bit3)
Floating-point invalid exception summary (FPSCR bit2)
Floating-point exception enabled summary (FPSCR bit 1)
Floating-point exception summary (FPSCR bit0)

Bit	Meaning
4	Set if bit 0 of the FPSCR is set. That is, this bit indicates whether any floating-point exception has occurred.
5	Set if bit 1 of the FPSCR is set. That is, this bit indicates whether any of the floating-point exceptions are enabled.
6	Set if bit 2 of the FPSCR is set. That is, this bit indicates whether an invalid exception has occurred for any reason.
7	Set if bit 3 of the FPSCR is set. That is, this bit indicates whether an overflow has occurred.

If you append a dot (.) to a floating-point instruction, its status will be recorded in the Condition Register as well as in the FPSCR. If you do not append a dot, the Condition Register will not reflect the result of that instruction.

Use Condition Register fields in conditional branch instructions. Several instructions allow you to store certain FPSCR bits in fields CR2 through CR4. After using one of these instructions, you then use a conditional branch instruction of the form

instr *field*, *address*

where *field* is the Condition Register field 2 through 4 and *address* is the address to branch to if the condition is true. Table 12-2 shows some commonly used PowerPC branch instructions. Examples of how to use the conditional branch instructions appear later in this chapter. For a complete list of conditional branch instructions, see the Motorola *PowerPC 601 RISC Microprocessor User's Manual.*

Table 12-2 Branch instructions using the Condition Register

Instruction	Description
`bta` *bit*, *address*	Branch to *address* if condition is true (*bit* = 1)
`blt` *field*, *address*	Branch to *address* if less than (bit 0 of *field* = 1)
`ble` *field*, *address*	Branch to *address* if less than or equal (bit 0 of *field* = 1 or bit 2 = 1)
`beq` *field*, *address*	Branch to *address* if equal (bit 2 of *field* = 1)
`bge` *field*, *address*	Branch to *address* if greater than or equal (bit 1 of *field* = 1 or bit 2 = 1)
`bgt` *field*, *address*	Branch to *address* if greater than (bit 1 of *field* = 1)
`bnl` *field*, *address*	Branch to *address* if not less than (bit 0 of *field* = 0)
`bne` *field*, *address*	Branch to *address* if not equal (bit 2 of *field* = 0)
`bng` *field*, *address*	Branch to *address* if not greater than (bit 1 of *field* = 0)
`bun` *field*, *address*	Branch to *address* if unordered (bit 3 of *field* = 1)
`bnu` *field*, *address*	Branch to *address* if not unordered (bit 3 of *field* = 0)

Inquiries: Class and Sign

As stated in Chapter 2, “Floating-Point Data Formats,” the result of a floating-point operation is either a normalized number, a denormalized number, a zero, a NaN, or an Infinity. This section describes how the class and sign of a floating-point number can be determined in PowerPC assembly language.

Floating-Point Result Flags and Condition Codes

FPSCR bits 15 through 19 are the floating-point result flags. Bit 15 is in FPSCR field 3, and bits 16 through 19 are in FPSCR field 4. For many instructions, FPSCR bits 15 through 19 specify the class and sign of the instruction’s result. For comparison instructions, bits 16 through 19 store the result of the comparison.

Bit	Meaning
15	The class descriptor. If this bit is set, the result is either a quiet NaN or a denormalized number, depending on the settings of bits 16 through 19.
16	< or < 0. For comparison operations, this bit is set if the first operand is less than the second operand. For other operations, this bit is set if the result is negative (< 0).
17	> or > 0. For comparison operations, this bit is set if the first operand is greater than the second operand. For other operations, this bit is set if the result is positive (> 0).
18	= or = 0. For comparison operations, this bit is set if the first operand is equal to the second operand. For other operations, this bit is set if the result is 0 (= 0).
19	Unordered or NaN. For comparison operations, this bit is set if either of the operands is a NaN. For other operations, this bit is set if the result is a NaN or an Infinity, depending on the value of bit 15.

Table 12-3 shows how bits 15 through 19 are interpreted, depending on whether the previous instruction was a comparison operation or not.

Table 12-3 Values for FPSCR bits 15 through 19

Bits 15–19	Result for comparisons	Result for other operations
00001	Unordered	Not applicable
00010	== (equal to)	+0
00100	> (greater than)	Positive normalized number
00101	Not applicable	+∞
01000	< (less than)	Negative normalized number

continued

Table 12-3 Values for FPSCR bits 15 through 19 (continued)

Bits 15–19	Result for comparisons	Result for other operations
01001	Not applicable	−∞
10001	Unordered	Quiet NaN
10010	== (equal to)	−0
10100	> (greater than)	Positive denormalized number
11000	< (less than)	Negative denormalized number

Example: Determining Class

To determine the class of a floating-point operation, copy the FPSCR bits to the Condition Register and then branch on the Condition Register field, as shown in Listing 12-1. To copy FPSCR bits to the Condition Register, use the `mcrfs` instruction, which has the form

```
mcrfs  DST, SRC
```

where *DST* is a 4-bit Condition Register field and *SRC* is an FPSCR field.

Listing 12-1 Determining the class of an assembler instruction result

```
   fadd  f0,f1,f2    # sets FPSCR bits 15–19 from f0
   mcrfs 2,3         # copy FPSCR bits 12–15 to CR2
   mcrfs 3,4         # copy FPSCR bits 16–19 to CR3
   # CR bits 11 – 15 are class and sign of f0
   bun   3,inf       # if bit 3 of CR3 is 1, result is
                     # Infinity or NaN
   beq   3,zero      # if bit 2 of CR3 is 1, result is zero
   blt   3,norm      # if bit 0 or 1 of CR3 is 1,
   bgt   3,norm      # result is a normalized or
                     # denormalized number

inf:
   bta   11,NaN      # if bit 11 is set, result = quiet NaN
                     # else result is an Infinity
```

```
norm:
   bta   11,denorm   # if bit 11 is set, result is denorm
                     # else result is norm

zero:
   # return class of zero

denorm:
   # return class of denormalized number
```

The `fadd` instruction, which adds two floating-point numbers, is one of the many floating-point instructions that set FPSCR bits 15 through 19 to the class and sign of its result. To read these FPSCR bits, Listing 12-1 copies them to the Condition Register using the `mcrfs` instruction. This instruction operates on 4-bit fields. Bits 15 through 19 are contained in two fields (3 and 4), so two separate `mcrfs` instructions are required to copy all pertinent bits to the Condition Register. Once the bits are copied, Condition Register fields 2 and 3 contain FPSCR fields 3 and 4, which means that Condition Register bits 11 through 15 reflect FPSCR bits 15 through 19. Next, the branch instructions test the values in the Condition Register and determine what type of result the `fadd` instruction had.

Setting the Rounding Direction

Bits 30 through 31 of the FPSCR specify the current rounding direction, as shown in Table 12-4. The section “Rounding Direction Modes” in Chapter 4, “Environmental Controls,” describes what the different rounding directions do.

Table 12-4 Rounding direction bits in the FPSCR

Mode	Bit 30	Bit 31
To nearest (default)	0	0
Toward zero	0	1
Upward	1	0
Downward	1	1

Bits 30 and 31 are in FPSCR field 7.

To set the rounding direction, use the `mtfsfi` instruction. It has the form

```
mtfsfi    DST, n
```

where *DST* is a 4-bit FPSCR field and *n* is an integer value to be copied into *DST.* Here are some examples.

```
mtfsfi   7,0   # set rounding direction to to-nearest
mtfsfi   7,1   # set rounding direction to toward-zero
mtfsfi   7,2   # set rounding direction to upward
mtfsfi   7,3   # set rounding direction to downward
```

Floating-Point Exceptions

The assembly-language numeric implementation contains the same five floating-point exception flags that are described in the IEEE standard. This section describes how to enable, disable, set, clear, and test these exception flags.

Exception Bits in the FPSCR

Table 12-5 summarizes the FPSCR bits that control floating-point exceptions. For each bit, it shows which FPSCR field contains that bit. Note that all of these bits, unless otherwise specified, are **sticky**; that is, once set, they stay set until you specifically clear them. For information on exactly what happens when a floating-point exception occurs, see the Motorola *PowerPC 601 RISC Microprocessor User's Manual.*

Table 12-5 Floating-point exception bits in the FPSCR

Exception	FPSCR field	Bit	Comment
All	0	0	Exception summary; set if any floating-point exception has occurred
	0	1*	Exception enable summary; set if any floating-point exception is enabled
Invalid	0	2*	Invalid exception summary; bits 7 through 12 or 21 through 23 tell why the exception occurred
	1	7	Signaling NaN
	2	8	∞ − ∞
	2	9	∞/∞
	2	10	0/0
	2	11	0 × ∞

Table 12-5 Floating-point exception bits in the FPSCR (continued)

Exception	FPSCR field	Bit	Comment
	3	12	Comparison operation produced invalid
	5	21	Software request produced invalid[†]
	5	22	Square root produced invalid[†]
	5	23	Convert-to-integer operation produced invalid
	6	24	Invalid exception enable / disable
Overflow	0	3	Overflow flag
	6	25	Overflow enable / disable
Underflow	1	4	Underflow flag
	6	26	Underflow enable / disable
Divide-by-zero	1	5	Divide-by-zero flag
	6	27	Divide-by-zero enable / disable
Inexact	1	6	Inexact flag
	3	13[*]	Fraction rounded
	3	14[*]	Fraction inexact
	6	28	Inexact enable / disable

[*] This field is not sticky; it applies only for the last instruction executed.
[†] Not implemented in MPC601.

Signaling and Clearing Floating-Point Exceptions

To signal or clear a floating-point exception explicitly, set or clear its bit in the FPSCR. For example, the following instructions signal an overflow exception and then clear that exception:

```
mtfsb1   3     # sets FPSCR bit 3 to 1, signaling overflow
mtfsb0   3     # clears FPSCR bit 3, so no overflow
```

These two instructions operate on individual FPSCR bits rather than on 4-bit FPSCR fields. The instruction `mtfsb1` sets the specified bit in the FPSCR to 1. The `mtfsb1` instruction shown here sets bit 3, which is the overflow exception flag; therefore this instruction signals that an overflow has occurred. Similarly, the `mtfsb0` instruction sets the specified FPSCR bit to 0 and therefore clears the overflow exception.

Enabling and Disabling Floating-Point Exceptions

To enable or disable a floating-point exception, set or clear its enable bit in the FPSCR.

Note

Disabling a floating-point exception does not mean that its flag will never be set. For the exact meaning of disabling a particular floating-point exception, see the Motorola *PowerPC 601 RISC Microprocessor User's Manual.* u

For example, the following instructions enable and then disable the overflow exception:

```
mtfsb1   25    # sets FPSCR bit 25; overflow enabled
mtfsb0   25    # clears FPSCR bit 25; overflow disabled
```

You can also use the following commands to enable and disable all floating-point exceptions at once:

```
mtfsi   6,0      # disables all floating-point exceptions
mtfsi   6,15     # enables all floating-point exceptions
```

As you can see from Table 12-1 on page 12-4, FPSCR field 6 contains all of the floating-point exception enable switches, so to enable or disable all floating-point exceptions at once, you need to set or clear this field. The `mtfsfi` instruction (described on page 12-10) copies a 16-bit signed integer value into an FPSCR field; so the first instruction shown here disables all floating-point exceptions by clearing all bits in field 6, and the second instruction enables all floating-point exceptions by setting all bits in field 6.

IMPORTANT

For the FPSCR exception enable bits to be valid, bit 20 or 23 of the Machine State Register must be set. For more information, see the Motorola *PowerPC 601 RISC Microprocessor User's Manual.* s

Testing for Floating-Point Exceptions

If you would like to see whether an exception occurred, test the Condition Register. Listing 12-2 checks the Condition Register to see if an exception has occurred and, if so, branches to a routine that determines the type of exception. It uses the `fadd.` form of the floating-point add instruction to copy the exception summary bits to Condition Register field 1. If the add instruction causes an exception, this example uses the `mcrfs` instruction (described on page 12-8) to copy the FPSCR fields containing floating-point exception flags to Condition Register fields 2 through 5 and then uses branch instructions to see which type of exception has occurred.

Listing 12-2 Testing for occurrence of floating-point exceptions

```
   fadd. f0,f1,f2 # f1 + f2 = f0. CR1 contains except.summary
   bta   4,error  # if bit 0 of CR1 is set, go to error
                  # bit 0 is set if any exception occurs
   .              # if clear, continue operation
   .
   .
error:
   mcrfs 2,1   # copy FPSCR bits 4–7 to CR field 2
               # now CR1 and CR2 (bits 6 through 10)
               # contain all exception bits from FPSCR
   bta   6,invalid   # CR bit 6 signals invalid
   bta   7,overflow  # CR bit 7 signals overflow
   bta   8,underflow # CR bit 8 signals underflow
   bta   9,divbyzero # CR bit 9 signals divide-by-zero
   bta   10,inexact  # CR bit 10 signals inexact

invalid:
   mcrfs 2,2   # copy FPSCR bits 8–11 to CR field 2
   mcrfs 3,3   # copy FPSCR bits 12–15 to CR field 3
   mcrfs 4,5   # copy FPSCR bits 20–23 to CR field 4
               # invalid bits are now CR bits 11–16 and bit 23

   # now do exception handling based on which invalid bit
   # is set

overflow:
   # do exception handling for overflow exception

underflow:
   # do exception handling for underflow exception

divbyzero:
   #do exception handling for the divide-by-zero exception

inexact:
   # do exception handling for the inexact exception
```

Saving and Restoring the Floating-Point Environment

To save and restore the state of the entire floating-point environment, use the `mffs` and `mtfsf` instructions.

The `mffs` instruction saves the FPSCR to a floating-point register. It has the form

```
mffs    DST
```

where *DST* is the floating-point register into which the FPSCR should be copied. For example, the instruction

```
mffs  f0
```

saves the current state of the FPSCR register in bits 32 through 63 of floating-point register F0. Bits 0 through 31 of register F0 are set to 1's.

To restore a floating-point environment that you have previously saved, use the `mtfsf` instruction. This instruction copies a 4-bit field from a floating-point register into an FPSCR field. It has the form

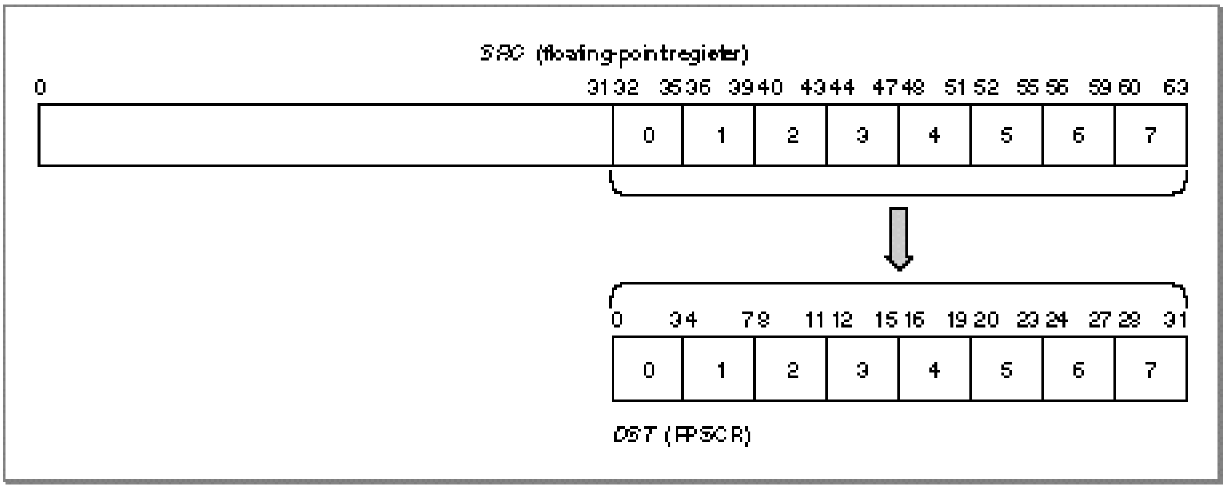
```
mtfsf      DST, SRC
```

where *DST* is a 4-bit FPSCR field and *SRC* is the floating-point register from which the field should be copied. The instruction assumes that the last half of the floating-point register *SRC* contains an FPSCR value. Thus, if you specify

```
mtfsf    3,f0
```

bits 44 through 47 of register F0 are copied into FPSCR field 3, bits 12 through 15. Figure 12-3 shows how the FPSCR fields map to a floating-point register.

Figure 12-3 *SRC* and *DST* fields for `mtfsf` instruction



Listing 12-3 saves the floating-point environment and then restores it.

Listing 12-3 Saving and restoring the floating-point environment

```
mffs  f10      # FPSCR copied into register f10

# other floating-point computations occur here

mtfsf 0,f10    # restore bits 0 and 3
mtfsf 1,f10    # restore bits 4 through 7
mtfsf 2,f10    # restore bits 8 through 11
mtfsf 3,f10    # restore bits 12 through 15
mtfsf 4,f10    # restore bits 16 through 19
mtfsf 5,f10    # restore bits 20 through 23
mtfsf 6,f10    # restore bits 24 through 27
mtfsf 7,f10    # restore bits 28 through 31
               # entire FPSCR now restored
```

CHAPTER 13

Assembly-Language Numeric Conversions

Contents

This chapter describes how you can use PowerPC assembly-language instructions to perform the conversions required by the IEEE standard (described in Chapter 5, "Conversions"). The assembler provides instructions that perform many of these conversions. The conversion instructions have two operands, both of which are floating-point registers. They are of the form

instr DST, *SRC*

and are interpreted as

DST ← *op SRC*

where *SRC* and *DST* are floating-point registers and *op* is some operation.

For each type of conversion, this chapter lists the assembly-language instructions you can use to perform that conversion and gives an example of how to use the instructions.

Conversions From Integer to Floating-Point Formats

No single instruction is available to convert an integer to floating-point format. However, you can perform this operation using the algorithm that follows. First, define the following constant:

```
kmagic: word 0x43300000,0x80000000
```

This constant must have an exponent of 52 after subtracting the bias for the double format (1023), and the lower half of the constant (bit 33) must begin with a 1. In the constant `kmagic` above, the first word (eight hexadecimal digits) corresponds to the exponent part and the last word corresponds to the integer part.

When you have an integer you want to convert, invert its sign, append the exponent part of the constant to the integer to be converted, and then load it into a floating-point register with the new exponent appended. Finally, subtract the floating-point constant from the newly formed floating-point integer. The following assembly code shows how this is done. The code fragment assumes that general-purpose register GPR0 contains the value 0 and that register GPR3 contains the value to be converted.

Listing 13-1 Converting a number from integer format to floating-point format

```
addis r1,r0,0x4330   # r1 contains 0x4300000
stw   r1,20000(r0)   # store exponent part for integer
xoris r3,r3,0x8000   # invert sign of integer
stw   r3,20004(r0)   # store fraction part for integer
                     # now all parts are in memory
lfd   f0,20000(r0)   # load integer in double format into f0
lfd   f1,kmagic(r0)  # load constant into f1
fsub  f0,f0,f1       # f0 contains converted integer
```

Conversions From Floating-Point to Integer Formats

To convert numbers in floating-point format to integer format, use one of two instructions:

`fctiw` Convert and round in current direction.

`fctiwz` Convert and round toward zero (truncate).

To convert double-format numbers to 32-bit integers, perform the following sequence of instructions:

```
lfd   f1,d(r2)    # load double float input into f1
fctiw f2,f1       # f2 is fixed 32-bit integer version of input
stfd  f2,d(r1)    # store f2 at location d + (r1)
lwz   r3,d+4(r1)  # r3 is fixed 32-bit integer version of input
```

To convert single-format numbers to 32-bit integers, perform the following sequence of instructions:

```
lfs   f1,d(r2)    # load single float input into f1
                  # input automatically converted to double format
fctiw f2,f1       # f2 is fixed 32-bit integer version of input
stfs  f2,d(r1)    # store f2 at location d + (r1)
lwz   r3,d+4(r1)  # r3 is fixed 32-bit integer version of input
```

To truncate double- or single-format numbers, replace `fctiw` in the above examples with `fctiwz`.

Note
The conversion instructions might raise floating-point exceptions. For more information, see the Motorola *PowerPC 601 RISC Microprocessor User's Manual.* u

Conversions From Single to Double Format

To convert a single floating-point number to double format, you simply load a single-format number into a floating-point register; the conversion takes place automatically. The following load instructions automatically convert single format to double. These instructions raise no floating-point exceptions and treat 0s, NaNs, and Infinities like any other value.

`lfs`	Load single format.
`lfsu`	Load single format and update.
`lfsux`	Load single format and update indexed.
`lfsx`	Load single format indexed.

For more information on the load instructions, see Chapter 11, "Introduction to Assembly-Language Numerics."

Conversions From Double to Single Format

To convert a double floating-point number to single format, either store the double number in single format (described in Chapter 11) or use the `frsp` instruction.

`frsp`	Convert double to single format.
`stfs`	Store in single format.
`stfsu`	Store in single format and update.
`stfsux`	Store in single format and update indexed.
`stfsx`	Store in single format indexed.

For store instructions, the conversion takes place automatically. The store instructions raise no floating-point exceptions and treat zeros, NaNs, and Infinities like any other value. The `frsp` instruction converts a double-format number to single format and then places it in the last half of a floating-point register. Use the `frsp` instruction immediately before using the single form of any arithmetic instruction. The following example performs single-precision addition on a number that has been converted to single format using the `frsp` instruction.

```
lfs   f1,d(r1)    #load single format number into f1
                  #conversion to double format is automatic
frsp  f1,f1       #f1 is now in single format
fadds f0,f1,f1    #so that it can be added as single format number
```

Note

The `frsp` instruction might raise floating-point exceptions. See the Motorola *PowerPC 601 RISC Microprocessor User's Manual* for more information. u

CHAPTER 14

Assembly-Language Numeric Operations

Contents

This chapter describes how you can perform comparison and arithmetic numeric operations using PowerPC assembly language. This chapter describes the following types of instructions:

- comparison
- arithmetic
- multiply-add
- move

It shows the format of these instructions and gives examples of use. For complete details on any of these instructions, see the Motorola *PowerPC 601 RISC Microprocessor User's Manual*. For operations that manipulate the floating-point environment, see Chapter 12, "Assembly-Language Environmental Controls." For operations that perform conversions, see Chapter 13, "Assembly-Language Numeric Conversions."

Comparison Operations

The assembler provides two floating-point comparison instructions:

`fcmpo`	Ordered comparison
`fcmpu`	Unordered comparison

The only difference is that the ordered comparison instruction generates an invalid exception if one of the input registers contains a NaN.

The comparison instructions have three operands. They are of the form

instr *DST*, *SRC1*, *SRC2*

DST A field in the Condition Register (0 through 7) into which the result of the comparison is placed.

SRC1, *SRC2* Two floating-point registers.

Comparison instructions are interpreted as

DST ← *SRC1* `compare` *SRC2*

The comparison instructions compare the contents of two floating-point registers and place the results of the comparison in a Condition Register field as well as in bits 16 through 19 (field 5) of the FPSCR. The results in the Condition Register and FPSCR are interpreted as follows:

Result	Meaning
0001	Unordered
0010	*SRC1* = *SRC2*
0100	*SRC1* > *SRC2*
1000	*SRC1* < *SRC2*

Use a conditional branch instruction after the comparison instruction to use the results of the comparison, as shown in the following example:

```
fcmpo    2,f0,f11 # compare f0 to f11 and put result in CR2
blt      2,addr1  # go to addr1 if bit 0 (<) of CR2 is 1
bgt      2,addr2  # go to addr2 if bit 1 (>) of CR2 is 1
beq      2,addr3  # go to addr3 if bit 2 (=) of CR2 is 1
bun      2,addr4  # go to addr4 if bit 3 (unordered) of CR2 is 1
```

Arithmetic Operations

PowerPC assembly language supports five of the seven IEEE arithmetic operations:

- add
- subtract
- multiply
- divide
- round-to-integer

Except for the round-to-integer operation, these operations may be performed by a variety of instructions. The instructions that perform arithmetic operations are divided into three categories: arithmetic instructions, multiply-add instructions, and move instructions. (`fctiw`, described in Chapter 13, “Assembly-Language Numeric Conversions,” performs the round-to-integer operation.)

Arithmetic Instructions

There are four arithmetic instructions:

`fadd`	Adds two floating-point values.
`fsub`	Subtracts two floating-point values.
`fmul`	Multiplies two floating-point values.
`fdiv`	Divides two floating-point values.

Note
These instructions might raise floating-point exceptions. See the Motorola *PowerPC 601 RISC Microprocessor User's Manual* for more information. u

Floating-point arithmetic instructions have three operands, all of which are floating-point registers. They are of the form

instr DST, *SRC1*, *SRC2*

Arithmetic instructions are interpreted as

DST ← *SRC1 op SRC2*

where *SRC1*, *SRC2*, and *DST* are floating-point registers and *op* is some operation.

Each of these instructions works on both single and double floating-point numbers. There are four versions of each instruction:

instr Perform operation specified by *instr*. Interpret data in floating-point registers as double format.

instr. Perform operation specified by *instr*. Interpret data in floating-point registers as double format. Record any exceptions raised in the Condition Register.

*instr*s Perform operation specified by *instr*. Interpret data in floating-point registers as single format.

*instr*s. Perform operation specified by *instr*. Interpret data in floating-point registers as single format. Record any exceptions raised in the Condition Register.

Note that all exceptions are always recorded in the FPSCR and are sometimes recorded in the Condition Register as well.

The following example adds two double floating-point numbers and stores the results:

```
lfd      f1,d(r1)    # load double number into register f1
lfd      f2,d(r2)    # load double number into register f2
fadd     f0,f1,f2    # f0 contains result
stfd     f0,d(r3)    # store result in double format
```

And the next example adds two single floating-point numbers and stores the results:

```
lfs      f1,d(r4)    # load single number into register f1
frsp     f1,f1       # stay single
lfs      f2,d(r5)    # load single number into register f2
frsp     f2,f2       # stay single
fadds.   f0,f1,f2    # result placed in f0 in single format
                     # CR1 reflects any exceptions
stfs     f0,d(r6)    # store result in single format
```

Multiply-Add Instructions

There are four multiply-add instructions:

`fmadd`	Perform multiply, add.
`fmsub`	Perform multiply, subtract.
`fnmadd`	Perform multiply, add, and negate.
`fnmsub`	Perform multiply, subtract, and negate.

Note

These instructions might raise floating-point exceptions. See the Motorola *PowerPC 601 RISC Microprocessor User's Manual* for more information. u

PowerPC assembly language provides the **multiply-add instructions** to perform more complex operations with at most a single roundoff error rather than the two potential roundoff errors that would result from performing the operations separately.

The multiply-add instructions take four operands, all of which are floating-point registers:

instr *DST*, *SRC1*, *SRC2*, *SRC3*

Multiply-add instructions are interpreted as

$$DST \leftarrow (SRC1 \times SRC2) \pm SRC3$$

where *SRC1*, *SRC2*, *SRC3*, and *DST* are floating-point registers.

Multiply-add instructions can take one of four forms:

instr Perform operation specified by *instr*. Interpret data in floating-point registers as double format.

instr`.` Perform operation specified by *instr*. Interpret data in floating-point registers as double format. Record any exceptions raised in the Condition Register.

instr`s` Perform operation specified by *instr*. Interpret data in floating-point registers as single format.

instr`s.` Perform operation specified by *instr*. Interpret data in floating-point registers as single format. Record any exceptions raised in the Condition Register.

Note that all exceptions are always recorded in the FPSCR and are sometimes recorded in the Condition Register as well.

The following example multiplies two double-format numbers, adds a third, and stores the result:

```
lfd      f1,d(r1)    # load double number into register f1
lfd      f2,d(r2)    # load double number into register f2
lfd      f3,d(r3)    # load double number into register f3
fmadd    f0,f1,f2,f3 # f0 = f1 × f2 + f3
stfd     f0,d(r4)    # store result as double format
```

The following example performs the same operations on single-format numbers:

```
lfs      f1,d(r5)    # load single number into register f1
frsp     f1,f1       # stay single
lfs      f2,d(r6)    # load single number into register f2
frsp     f2,f2       # stay single
lfs      f3,d(r7)    # load single number into register f3
frsp     f3,f3       # stay single
fmadds.  f0,f1,f2    # f0 = f1 × f2 + f3
                     # f0 contains single format number
                     # CR1 reflects any exceptions
stfs     f0,d(r8)    # store result in single format
```

Move Instructions

There are four move instructions:

`fabs`	Move absolute value of register.
`fmr`	Move register value.
`fneg`	Move negative value of register.
`fnabs`	Move negative absolute value of register.

Move instructions perform sign manipulations while copying a value from one floating-point register to another. Because they manipulate only the sign bit, they generate no floating-point exceptions. They take two operands, both of which are floating-point registers. They are of the form

instr *DST*, *SRC*

Floating-point move instructions are interpreted as

DST ← *op SRC*

where *SRC* and *DST* are floating-point registers and *op* is some operation that is performed on the contents of *SRC*.

Note that you may copy a value from a register into the same register. For example:

```
fneg  f1,f1    # f1 has just been negated
```

Transcendental and Auxiliary Functions

PowerPC assembly language does not directly support any of the IEEE auxiliary functions or the transcendental functions listed in this book. If you are writing a numerics application in assembly language, you can access the routines in the C library MathLib to perform these operations, provided you set up the stack frame properly. For information on how to set up the stack frame, see the book *Assembler for Macintosh With PowerPC*.

Appendixes

APPENDIX A

SANE Versus PowerPC Numerics

This appendix describes how PowerPC Numerics differs from the Standard Apple Numerics Environment (SANE) and tells you how to port programs that use SANE features so that they use PowerPC Numerics features instead. SANE is the numerics environment used on 680x0-based Macintosh computers. If you have written programs that perform floating-point computations for a 680x0-based Macintosh computer, that program uses SANE features. Unlike PowerPC Numerics, SANE is not compliant with the recommendations in the FPCE technical report. Compliance with the FPCE report allows a higher level of portability.

If you run a 680x0 application on a PowerPC processor-based Macintosh computer, it uses SANE instead of the PowerPC Numerics environment unless you recompile the program with a PowerPC compiler. If you want to recompile a program written for the 680x0-based Macintosh computer, you might have to modify some of your code.

Read this chapter if you are familiar with SANE and you want to know how PowerPC Numerics compares with SANE. The first section lists the differences between SANE and PowerPC Numerics. The last section provides some suggestions for porting your code.

Comparison of SANE and PowerPC Numerics

This section goes chapter by chapter through Part 1 of the *Apple Numerics Manual*, second edition, and tells where the two environments are alike and where they differ.

Floating-Point Data Formats

The single and double data formats supported by PowerPC Numerics are identical to the single and double data formats supported by SANE. PowerPC Numerics adds the double-double format not supported in SANE. PowerPC Numerics does not support the SANE floating-point formats comp (**integral value**) and 80-bit (and 96-bit) extended.

Conversions

PowerPC Numerics converts any floating-point format or integer format to any other floating-point format. SANE supports only conversions to and from the extended data format because it performs all floating-point operations in extended precision.

Conversions between binary and decimal in SANE are accurate up to a certain number of decimal digits for each floating-point data format. All conversions in PowerPC Numerics except conversions to or from double-double are correctly rounded.

Expression Evaluation

SANE uses the extended data format as the minimum evaluation format for all floating-point operations. All operations are evaluated with the greatest amount of precision possible, which ensures against midexpression overflow and underflow.

PowerPC Numerics does not specify one evaluation method but strongly recommends a single or double minimum evaluation format with widest-need evaluation. This method permits all expressions to be evaluated in as wide a format as is necessary without forcing a wider format on expressions that could be done more quickly and as accurately with a narrower format.

Note that with PowerPC Numerics, you are not ensured against midexpression overflow and underflow as you are with SANE. For example, suppose you have the following expression:

```
double d1, d2, d3, result;

d1 = d2 = d3 =  1.7E308;   /* maximum number in double */
result = (d1 + d2) / d3;
```

With PowerPC Numerics, the expression `d1 + d2` will overflow the double format, thus producing +∞. Infinity divided by the variable `d3` will still be +∞, and so the variable `result` will be assigned +∞. With SANE, `d1`, `d2`, and `d3` are converted to extended format. The expression `d1 + d2` will not overflow the extended format, and so the variable `result` will be assigned the value 2.

Infinities, NaNs, and Denormalized Numbers

Infinities, NaNs, and denormalized numbers are represented and used identically in SANE and PowerPC Numerics.

Arithmetic and Comparison Operations

SANE and PowerPC Numerics support the same seven basic arithmetic operations (add, subtract, multiply, divide, square root, remainder, and round-to-integer). SANE has only one version of the remainder and round-to-integer functions. PowerPC Numerics has two versions of the remainder function and several round-to-integer functions.

Note

SANE’s square root, remainder, and round-to-integer functions return type `extended` and take type `extended` as input. PowerPC Numerics uses type `double` instead. u

Both SANE and PowerPC Numerics support the following comparison operators: <, <=, >=, >, ==, and !=. All other comparison operators shown in Table 6-1 on page 6-4 in this book are not supported by SANE.

Environmental Controls

SANE and PowerPC Numerics support the same rounding direction modes and the same floating-point exception flags.

SANE supports dynamic rounding precision modes because it performs all operations in extended and is therefore required by IEEE to support dynamic rounding precision modes. PowerPC Numerics does not support rounding precision modes.

SANE supports halts for each of the five floating-point exceptions. PowerPC Numerics does not currently support halts, although it might in the future.

Transcendental (Elementary) Functions

In SANE, all transcendental functions are in extended format. That is, all of them take type `extended` for floating-point input and all of them return type `extended`. In PowerPC Numerics, all transcendental functions take double for floating-point input and return type `double`. Some of the functions have a version that performs the same operation in double-double precision.

SANE supports a subset of the transcendental functions that PowerPC Numerics supports. The functions not supported by SANE are

`erf`	`erfc`	`fdim`
`fmax`	`fmin`	`gamma`
`lgamma`	`nearbyint`	`rinttol`
`round`	`roundtol`	`trunc`

Of the functions supported by both SANE and PowerPC Numerics, a few are implemented differently in the two environments. See the section “Differences in Transcendental Functions” on page A-5 for details.

Porting SANE to PowerPC Numerics

If you have a program that is written to take advantage of SANE features, you might want to port it to the PowerPC processor to take advantage of the increased speed. This section provides tips on how to do so.

Perform the following steps to be sure that your program will run on both 680x0-based and PowerPC processor-based Macintosh computers:

1. Replace all uses of type `comp` with type `double` or `long int`.
2. Replace `sane.h` and `math.h` with `fp.h` and `fenv.h`.
3. Replace uses of `extended` with `double_t` or, if this is not possible, with `long double`.
4. Replace SANE-specific functions with their MathLib equivalents. SANE-specific functions include the functions listed as implemented differently in MathLib in the section "Differences in Transcendental Functions" on page A-5, all class and sign inquiry functions, and all environmental control functions.

The following sections guide you through these four steps.

Replacing Variables of Type comp

The first step in porting a SANE program is to remove uses of the data type `comp`. The type `comp` is a floating-point type with 64 bits of precision. In SANE, type `comp` is automatically converted to extended format whenever an expression is evaluated, just like every other SANE data format. In other words, `comp` is a floating-point type disguised as an integer type. In most cases you can replace type `comp` with type `double`, which provides 53 bits of precision. If your `comp` variables require greater than 53 bits of precision, you might need to write your own integer arithmetic package.

Using MathLib Instead of the SANE Library

The next step in porting a SANE program is to use the header files `fp.h` and `fenv.h`. The files `fp.h` and `fenv.h` replace `sane.h` and `math.h`. All of the transcendental functions declared in `sane.h` are now declared in `fp.h`, and most of them work exactly the same way in the two environments. If your program includes the header file `math.h` instead of `sane.h`, you should replace it with `fp.h` as well. The `fp.h` file declares all of the functions and macros declared in the ANSI header file `math.h` plus some additional ones.

Be aware of the differences in function prototypes in the files `sane.h` and `fp.h`. If your program currently uses `sane.h`, the declarations for transcendental functions look like this:

`extended` *func_name* `(extended` *func_params*`);`

In other words, all transcendental functions in `sane.h` are type `extended` and take type `extended` as arguments. These declarations mean that you can pass any floating-point type to a transcendental function without losing precision.

In `fp.h`, the typical transcendental function declaration has the form

`double_t` *func_name* `(double_t` *func_params*`);`

The `double_t` type changes definition based on which processor the program is run. For the PowerPC processor, `double_t` is defined to be type `double`. For the 680x0 processor, `double_t` is defined to be type `extended`. Therefore, when you change from using `sane.h` to using `fp.h`, your program will compile on both the 680x0 and the PowerPC processors and there will be no change in the way your program runs on the 680x0. For more information on the `double_t` type, see "Portable Declarations" on page A-9.

In some cases, a numeric function also has a `long double` implementation in MathLib. The declarations of the `long double` implementations are in `fp.h` and have the form

```
long double func_namel (long double func_params);
```

See the function descriptions in Part 2 of this book to find out if a function you are using has a `long double` implementation. If it does, you should examine the types of the parameters you are passing to that function and you should examine the return values. If a function parameter or return value requires more than 53 bits of precision, you may need to use the `long double` implementation of the function when it runs on a PowerPC processor. To do this, you simply add the letter *l* to the function call.

Replacing Extended Format Variables

When changing extended variables, first change all variables that are declared as `extended` to type `double_t`. For the 680x0 processor, `double_t` is defined as `extended`. For the PowerPC processor, `double_t` is defined as `double`. Once you make this change, your program runs with no changes on the 680x0 processor but now also runs on the PowerPC processor. Next, you need to examine each `double_t` variable to see if it will overflow on the PowerPC processor. If the variable requires more than 53 bits of precision, change its declaration to `long double`.

Using MathLib Functions

As mentioned previously, PowerPC Numerics (specifically, the MathLib library) provides a superset of the functions that SANE provides. In most cases you don't need to make any changes to your existing calls to the SANE library. However, there are a few transcendental functions that have a different implementation in MathLib. Also, the names have changed for the class and sign inquiries and floating-point environmental controls.

Differences in Transcendental Functions

The following transcendental functions are implemented differently in MathLib than in the SANE library:

- The `copysign` function does not follow the IEEE standard in SANE, which reverses the order of the function's parameters. PowerPC Numerics follows the parameter order described in the IEEE standard.

- The `exp1` function in SANE is named `expm1` in PowerPC Numerics.
- The `ipower` function is replaced with the `pow` function in PowerPC Numerics.
- The `log1` function in SANE is named `log1p` in PowerPC Numerics.
- The nextafter functions in SANE are `nextfloat`, `nextdouble`, and `nextextended`. In PowerPC Numerics, they are `nexafterf`, `nextafterd`, and `nextafterl` for `float`, `double`, and `long double`, respectively.
- The `nan` function in SANE takes a character parameter, but the PowerPC Numerics `nan` function takes a character string parameter.
- The SANE `pi` function is replaced with the constant `pi`, the SANE `inf` function is replaced with the constant `INFINITY`, and the `NAN` constant remains the same.
- The `pow` function behaves differently in the two environments. For example, in SANE `pow(NAN,0)` returns a NaN, whereas in PowerPC Numerics, `pow(NAN,0)` returns a 1.
- The `remainder` function in SANE takes three parameters, the last one being a return value. The PowerPC Numerics `remainder` function takes two parameters. The `remquo` function is analogous to the SANE `remainder` function.
- The `scalb` function does not follow the IEEE standard in SANE, which reverses the order of the function's parameters. PowerPC Numerics follows the parameter order described in the IEEE standard.

Differences in Class and Sign Inquiries

The class and sign inquiry functions declared in `sane.h` are not implemented in MathLib. Instead, MathLib provides a set of macros that perform the same actions. Table A-1 shows the declarations in `sane.h` on the left and the corresponding declaration in the MathLib header file `fp.h` on the right.

Table A-1 Class and sign inquiries in SANE versus MathLib

`sane.h` declaration	`fp.h` declaration
	`enum NumberKind {`
`#define SNAN        0`	`FP_SNAN = 0,`
`#define QNAN        1`	`FP_QNAN,`
`#define INFINITE    2`	`FP_INFINITE,`
`#define ZERONUM     3`	`FP_ZERO,`
`#define NORMALNUM   4`	`FP_NORMAL,`
`#define DENORMALNUM 5`	`FP_SUBNORMAL };`
`typedef short numclass;`	
`numclass classfloat (extended x);`	`#define fp_classify(x)`[*]
`numclass classdouble(extended x);`	
`numclass classcomp(extended x);`	
`numclass classextended(extended x);`	
`long signnum (extended x);`	`#define signbit(x)`

[*] The `fpclassify` macro returns a long integer.

Differences in Environmental Controls

MathLib's environmental control functions are declared in the header file `fenv.h`. They affect only rounding direction modes and floating-point exceptions, and they are different from the functions that perform the same tasks in the SANE library.

If the SANE program uses rounding precision modes, you must remove this code to run it on the PowerPC processor. The PowerPC processor almost always uses less precision than SANE when evaluating expressions, so this should not be a problem. See Chapter 3, "Expression Evaluation," for details.

If the SANE program uses halts, you need to replace them with your own exception handling routines.

Replace the floating-point environmental access function or macro on the left side of Table A-2 with the corresponding function or macro on the right side. If your compiler supports the environmental access switch described in Appendix D, "FPCE Recommendations for Compilers," you must turn the switch on before using any of the functions or macros from Table A-2.

Table A-2 Environmental access functions in SANE versus MathLib

`sane.h` declaration	`fenv.h` declaration
#define INVALID 1 #define UNDERFLOW 2 #define OVERFLOW 4 #define DIVBYZERO 8 #define INEXACT 16	#define FE_INEXACT 0x02000000 #define FE_DIVBYZERO 0x04000000 #define FE_UNDERFLOW 0x08000000 #define FE_OVERFLOW 0x10000000 #define FE_INVALID 0x20000000
#define IEEEDEFAULTENV	#define FE_DFL_ENV &_FE_DFL_ENV
typedef short exception;	typedef long int fexcept_t;
typedef short environment	typedef long int fenv_t;
#define TONEAREST 0 #define UPWARD 1 #define DOWNWARD 2 #define TOWARDZERO 3	#define FE_TONEAREST 0x00000000 #define FE_TOWARDZERO 0x00000001 #define FE_UPWARD 0x00000002 #define FE_DOWNWARD 0x00000003
typedef short rounddir;	—

void setexception(exception e,
long s);
	int fesetexcept(const fexcept_t
	*flagp, int excepts);
	int feclearexcept(int excepts);
	int feraiseexcept(int excepts);

long testexception(exception e);	int fetestexcept(int excepts);
void setround (rounddir r);	int fesetround(int round);
rounddir getround(void);	int fegetround(void);

continued

Table A-2 Environmental access functions in SANE versus MathLib (continued)

`sane.h` declaration	`fenv.h` declaration
`void setenvironment(environment e);`	`void fesetenv(const fenv_t *envp);`
`void getenvironment(environment *e);`	`void fegetenv(fenv_t *envp);`
`void procentry(environment *e);`	`int feholdexcept(fenv_t *envp);`*
`void procexit(environment e);`	`void feupdateenv(const fenv_t *envp);`

* The `feholdexcept` function, although it replaces the `procentry` SANE function, affects only the exception flags. It does not affect the rounding direction.

Listing A-1 is a C code fragment that runs on both the 680x0 and PowerPC processors. It performs the `pow` function, tests for the occurrence of the inexact exception, and prints the results.

Listing A-1 Using environmental controls in SANE and PowerPC Numerics

```
   double_t x, y, result;/* double on PowerPC,extended on 680x0 */
#ifdef __SANE__          /* 680x0 processor */
   exception fp_inexact;
#else                    /* PowerPC processor */
   fexcept_t fp_inexact;
#endif

#ifdef __SANE__          /* 680x0 processor */
   setenvironment(IEEEDEFAULTENV);
#else                    /* PowerPC processor */
   fesetenv(FE_DFL_ENV);
#endif

   result = pow(x, y);

#ifdef __SANE__          /* 680x0 processor */
   fp_inexact = testexception (INEXACT);
#else                    /* PowerPC processor */
   fp_inexact = fetestexcept (FE_INEXACT);
#endif

   printf ("pow(%g,%g) = %g\t", x, y, result);
   if (fp_inexact)
      printf ("INEXACT\n");
```

Compatibility Tools in MathLib

This section describes some tools provided in MathLib that help with compatibility between two environments. The tools include type definitions that help you make efficient, portable variable declarations and macros that are defined differently on the two architectures.

Portable Declarations

MathLib defines two floating-point type definitions, `float_t` and `double_t`, in the header file `Types.h`. If you define a variable to be `float_t` or `double_t`, it means "use the most efficient floating-point type for this architecture." Table A-3 shows the definitions for `float_t` and `double_t` on PowerPC architecture compared with 680x0 architecture.

Table A-3 `float_t` and `double_t` definitions

Architecture	**`float_t`**	**`double_t`**
PowerPC	`float`	`double`
680x0	`long double`	`long double`

The PowerPC architecture is based on the IEEE double format. The most natural format for computations is double, but the architecture allows computations in single format as well. Therefore, `float_t` is defined to be `float` (single precision) and `double_t` is defined to be `double` for the PowerPC architecture. The 680x0 architecture is based on the extended format and performs all computations in extended format regardless of the type of the operands. Therefore, `float_t` and `double_t` are both `long double` (extended precision) for the 680x0 architecture.

If you declare a variable to be type `double_t` and you compile the source code as a PowerPC application, the variable is double format. If you recompile the same source code as an 680x0 application, the variable is extended format.

If your compiler is FPCE-compliant, it also supports the pragmas that allow the most efficient floating-point type to be used for function return values, parameters, and local variables. See Appendix D, "FPCE Recommendations for Compilers," for more information on these pragmas.

Macros

You might find the following macros useful to isolate 680x0-specific code from PowerPC-specific code:

Macro	Description
`__SANE__`	Defined if `sane.h` is used
`__FP__`	Defined if `fp.h` is used
`LONG_DOUBLE_SIZE`	Returns the size in bytes of `long double` on the processor on which the program is run
`DOUBLE_SIZE`	Returns the size in bytes of `double` on the processor on which the program is run
`DECIMAL_DIG`	Returns the maximum size in digits of a decimal number that can be converted to binary

APPENDIX B

Porting Programs to PowerPC Numerics

This appendix contains information of interest to programmers who are porting programs from a non-Macintosh computer to run on a PowerPC processor-based Macintosh computer using PowerPC Numerics. If you are such a programmer and you think you are getting errors because of differences in numerics, you should read this appendix.

Porting applications to run in the PowerPC Numerics environment is easier than porting to other computers. Expressions that produce good results on other computers usually give at least as good results using PowerPC Numerics.

Note

If you are porting a program that uses SANE, read Appendix A, “SANE Versus PowerPC Numerics,” instead of this appendix. u

Semantics of Arithmetic Evaluation

When you translate programs from one language to another, be aware of the hidden pitfalls in translation. For example, an operation in one language might have similar syntax to an operation in another language without being similar semantically. Here’s an example of similar functions with different syntaxes:

- Fortran, `SIGN(A,B)` (two operands)
- BASIC, `SIGN(A)` (one operand)

Languages can also differ in how they treat mixed integers and reals. For example, Fortran truncates integer quotients to integers, so `3/7 = 0` (you have to write 3.0/7.0 to obtain a fraction). The programmer translating must be aware that the results of such expressions depend on the language used.

Languages also differ in how they convert from a real number to an integer. For example, in Fortran, assigning a floating-point value to an integer rounds toward zero.

Here are the operations used to truncate a real number to an integer in three languages:

- C: assignments and casts
- Fortran: `AINT`, `INT`
- Pascal: `Trunc`

Mixed Formats

On certain computers, the formats for single and double are identical except for their length. On those machines, for arguments passed by address, a calling routine can store data in one format and a called routine can read data in another format without apparent error.

If you have a program that exploits this confusion, you'll have to revise it before you can run it on a machine that uses PowerPC Numerics. (Type checking is of no help here; if the discrepancy was such that type checking could detect it, the original compiler would have caught it.)

Floating-Point Precision

Floating-point precision may differ from the original machine to the target machine.

Some computers have floating-point formats that have a wider range than the current PowerPC Numerics formats. Wider formats include the VAX H format, the IBM Q format, and the HP quad format. Programs use these wide formats for computation involving input data from a narrower format to minimize the occurrence of overflow and underflow and to preserve accuracy. The double-double data format provides enough precision to preserve accuracy; but it offers no greater range than the double format, so it will not protect against overflow and underflow. Keep in mind that problems may arise when a program uses formats wider than double-double.

CDC and Cray computers have a single format that is wider than IEEE single and a double format that is wider than IEEE double format. When porting code from those machines, you should consider changing type declarations from single to double format.

The Rules of Evaluation

Each computer uses different rules of evaluation. Here are three reasonable ones:

- Rule 1: Round the result to the wider of the two operand formats.
- Rule 2: Round the result to the widest available format.
- Rule 3: Round the result to the widest format in the expression.

Rule 1 is instant rounding. It is the rule on computers having many registers the same width as memory. This rule has been used by IBM and CDC Fortran since 1963. It is not part of the Fortran standard, though it is often thought to be.

Rule 2 is what SANE does by evaluating in extended precision. Other machines using this approach include the PDP-11C (using double precision) and floating-point coprocessors such as the 8087 and the MC68881. This approach does not take best advantage of machines with separate processing units for each floating-point format.

Rule 3 is what PowerPC Numerics does and is the way you do it when computing by hand. It was the rule in Fortran until 1963. By this rule, if you see an expression with mixed precision, you assume the user wants the widest visible precision.

With PowerPC Numerics, you can write code to simulate any of these rules. To simulate rule 1, use separate assignments when computing subexpressions. To simulate rule 2, convert all operands to double-double format before performing an expression.

For transported code, either you have to understand the programmer's tricks or you have to mimic the way rounding works on the programmer's machine. With PowerPC Numerics, you can set the rounding direction to mimic other machines.

The Invalid Exception

Many computers used to stop on an invalid operation, such as 0/0. Programmers have made the best of this and not bothered to test in advance for values that could cause an invalid operation. It is better to stop than to give a plausible but incorrect answer.

When a program written that way runs on PowerPC Numerics, it produces a NaN where it formerly would have stopped. The NaN might cause the program to take an unplanned branch and thus produce an erroneous answer. Because the program does not test for invalid operations, the user will not know whether the answers the program finally delivers have been influenced by exceptional events that formerly would have stopped the computer.

Programs sometimes contain code that depends on an ill-documented effect or on one that varies from machine to machine. If you have inherited such a program and you do not know what it does about exceptional conditions, here are some possible strategies:

- Insert tests on operands that could cause invalid operations.
- Change the program to make sure that NaNs propagate as NaNs rather than as plausible answers.
- After evaluations, add code to test the invalid flag and deliver a meaningful result or message and then clear the flag.

If you have a program with code you can't change and you distrust the results it gives when invalid operations occur, you should set up tests that halt programming on those invalid operations and set the environment to simulate the environment in which the program was designed to run.

MathLib Header Files

This appendix shows the contents of the two MathLib header files fp.h and fenv.h. You can use this appendix to see where and how a MathLib function is defined and to see which transcendental functions are available in MathLib.

Floating-Point Header File (fp.h)

The header file fp.h defines a collection of numerical functions designed to facilitate a wide range of numerical programming. It is modeled after the FPCE technical report. This file declares many functions in support of numerical programming. It provides a superset of math.h and sane.h functions. Some functionality previously found in sane.h on 680x0-based Macintosh computers and not in the FPCE fp.h can be found in this fp.h under the heading __NOEXTENSIONS__.

Constants

```
#ifndef  __FP__
#define  __FP__

/* efficient types are included in Types.h. */
#ifndef  __TYPES__
#include <Types.h>
#endif

#ifdef   powerc
#define           LONG_DOUBLE_SIZE     16
#elif    mc68881
#define           LONG_DOUBLE_SIZE     12
#else
#define           LONG_DOUBLE_SIZE     10
#endif      /* powerc */

#define           DOUBLE_SIZE           8

#define           HUGE_VAL             __inf()
#define           INFINITY             __inf()
#define           NAN                  nan("255")
```

```
/* the macro DECIMAL_DIG is obtained by satisfying the constraint that the
   conversion from double to decimal and back is the identity function. */

#ifdef   powerc
#define           DECIMAL_DIG          36
#else
#define           DECIMAL_DIG          21
#endif   /* powerc */

#define      SIGDIGLEN      36           /* significant decimal digits */
#define      DECSTROUTLEN   80           /* max length for dec2str output */
#define      FLOATDECIMAL   ((char)(0))
#define      FIXEDDECIMAL   ((char)(1))
```

Inquiry Macros

```
#define  fpclassify  (x)   ((    sizeof (x) == LONG_DOUBLE_SIZE)     ?  \
                                 __fpclassify  (x)                    :  \
                                 (sizeof (x) == DOUBLE_SIZE)          ?  \
                                 __fpclassifyd (x)                    :  \
                                 __fpclassifyf (x))

/* isnormal is nonzero if and only if the argument x is normalized. */

#define  isnormal    (x)   ((    sizeof (x) == LONG_DOUBLE_SIZE)     ?  \
                                 __isnormal (x)                       :  \
                                 (sizeof (x) == DOUBLE_SIZE)          ?  \
                                 __isnormald (x)                      :  \
                                 __isnormalf (x))

/* isfinite is nonzero if and only if the argument x is finite. */

#define  isfinite    (x)   ((    sizeof (x) == LONG_DOUBLE_SIZE)     ?  \
                                 __isfinite (x)                       :  \
                                 ( sizeof (x) == DOUBLE_SIZE)         ?  \
                                 __isfinited (x)                      :  \
                                 __isfinitef (x))

/* isnan is nonzero if and only if the argument x is a NaN. */
```

```
#define  isnan       (x)   ((    sizeof (x) == LONG_DOUBLE_SIZE)     ?  \
                                 __isnan (x)                         :  \
                                 (sizeof (x) == DOUBLE_SIZE)         ?  \
                                 __isnand (x)                        :  \
                                 __isnanf (x))

/* signbit is nonzero if and only if the sign of the argument x is
   negative. This includes NaNs, infinities and zeros. */

#define  signbit     (x)   ((    sizeof (x) == LONG_DOUBLE_SIZE)     ?  \
                                 __signbit (x)                       :  \
                                 (sizeof (x) == DOUBLE_SIZE)         ?  \
                                 __signbitd (x)                      :  \
                                 __signbitf (x))
```

Data Types

```
enum NumberKind
{
   FP_SNAN = 0,                                 /* signaling NaN */
   FP_QNAN,                                     /* quiet NaN */
   FP_INFINITE,                                 /* + or – infinity */
   FP_ZERO,                                     /* + or – zero */
   FP_NORMAL,                                   /* all normal numbers */
   FP_SUBNORMAL                                 /* denormal numbers */
};

typedef short relop;

enum
{
   GREATERTHAN = ((relop) (0)),
   LESSTHAN,
   EQUALTO,
   UNORDERED
};

struct decimal
{
   char sgn;                        /* sign 0 for +, 1 for – */
   char unused;
   short exp;                       /* decimal exponent */
   struct
   {
      unsigned char length;
      unsigned char text[SIGDIGLEN];      /* significant digits */
      unsigned char unused;
   } sig;
};
typedef struct decimal decimal;

struct decform
{
   char style;                        /* FLOATDECIMAL or FIXEDDECIMAL */
   char unused;
   short digits;
};
typedef struct decform decform;

extern const double_t pi;
```

Functions

Trigonometric Functions

```
double_t cos             (double_t x);
double_t sin             (double_t x);
double_t tan             (double_t x);
double_t acos            (double_t x);  /* argument is in [0,pi] */
double_t asin            (double_t x);  /* argument is in [-pi/2,pi/2] */
double_t atan            (double_t x);  /* argument is in [-pi/2,pi/2] */

#ifdef powerc
long double cosl         (long double x);
long double sinl         (long double x);
long double tanl         (long double x);
long double acosl        (long double x); /*argument is in [0,pi]*/
long double asinl        (long double x); /*argument is in [-pi/2,pi/2]*/
long double atanl        (long double x); /*argument is in [-pi/2,pi/2]*/
#endif /* powerc */
```

```
double_t atan2              (double_t y, double_t x);

#ifdef powerc
long double atan2l          (long double y, long double x);
#endif /* powerc */
```

Hyperbolic Functions

```
double_t cosh               (double_t x);
double_t sinh               (double_t x);
double_t tanh               (double_t x);
double_t acosh              (double_t x);
double_t asinh              (double_t x);
double_t atanh              (double_t x);

#ifdef powerc
long double coshl           (long double x);
long double sinhl           (long double x);
long double tanhl           (long double x);
long double acoshl          (long double x);
long double asinhl          (long double x);
long double atanhl          (long double x);
#endif /* powerc */
```

Exponential Functions

```
double_t exp                (double_t x);

#ifdef powerc
long double expl            (long double x);
#endif /* powerc */

double_t expm1              (double_t x);

#ifdef powerc
long double expm1l          (long double x);
#endif /* powerc */
```

```
double_t exp2               (double_t x);
double_t frexp              (double_t x, int *exponent);
double_t ldexp              (double_t x, int n);
double_t log                (double_t x);

#ifdef powerc
long double exp2l           (long double x);
long double frexpl          (long double x, int *exponent);
long double ldexpl          (long double x, int n);
long double logl            (long double x);
#endif /* powerc */

double_t log2               (double_t x);

#ifdef powerc
long double log2l           (long double x);
#endif /* powerc */

double_t log1p              (double_t x);
double_t log10              (double_t x);

#ifdef powerc
long double log1pl          (long double x);
long double log10l          (long double x);
#endif /* powerc */

double_t logb               (double_t x);

#ifdef powerc
long double logbl           (long double x);
#endif /* powerc */

long double modfl           (long double x, long double *iptrl);
double modf                 (double x, double *iptr);
float modff                 (float x, float *iptrf);
```

```
double_t scalb              (double_t x, long int n);

#ifdef powerc
long double scalbl          (long double x, long int n);
#endif      /* powerc */
```

Power and Absolute Value Functions

```
double_t fabs               (double_t x);

#ifdef powerc
long double fabsl           (long double x);
#endif /* powerc */

double_t hypot              (double_t x, double_t y);
double_t pow                (double_t x, double_t y);
double_t sqrt               (double_t x);

#ifdef powerc
long double hypotl          (long double x, long double y);
long double powl            (long double x, long double y);
long double sqrtl           (long double x);
#endif /* powerc */
```

Gamma and Error Functions

```
double_t erf                (double_t x);           /* the error function */
double_t erfc               (double_t x); /* complementary error function */
double_t gamma              (double_t x);

#ifdef powerc
long double erfl            (long double x);        /* the error function */
long double erfcl           (long double x);/*complementary error function*/
long double gammal          (long double x);
#endif /* powerc */

double_t lgamma             (double_t x);

#ifdef powerc
long double lgammal         (long double x);
#endif /* powerc */
```

Nearest Integer Functions

```
double_t ceil                (double_t x);
double_t floor               (double_t x);

#ifdef powerc
long double ceill            (long double x);
long double floorl           (long double x);
#endif /* powerc */

double_t rint                (double_t x);

#ifdef powerc
long double rintl            (long double x);
#endif /* powerc */

double_t nearbyint           (double_t x);

#ifdef powerc
long double nearbyintl       (long double x);
#endif /* powerc */

long int rinttol             (double_t x);

#ifdef powerc
long int rinttoll            (long double x);
#endif /* powerc */

double_t round               (double_t x);

#ifdef powerc
long double roundl           (long double x);
#endif /* powerc */

long int roundtol            (double_t round);

#ifdef powerc
long int roundtoll           (long double round);
#endif /* powerc */
```

```
double_t trunc              (double_t x);

#ifdef powerc
long double truncl          (long double x);
#endif /* powerc */
```

Remainder Functions

```
double_t fmod               (double_t x, double_t y);
double_t remainder          (double_t x, double_t y);
double_t remquo             (double_t x, double_t y, int *quo);

#ifdef powerc
long double remainderl      (long double x, long double y);
long double remquol         (long double x, long double y, int *quo);
#endif /* powerc */
```

Auxiliary Functions

```
double_t copysign           (double_t x, double_t y);

#ifdef powerc
long double copysignl       (long double x, long double y);
#endif /* powerc */

long double nanl            (const char *tagp);
double nan                  (const char *tagp);
float nanf                  (const char *tagp);
long double nextafterl      (long double x, long double y);
double nextafterd           (double x, double y);
float nextafterf            (float x, float y);
```

Maximum, Minimum, and Positive Difference Functions

```
double_t fdim               (double_t x, double_t y);

#ifdef powerc
long double fdiml           (long double x, long double y);
#endif
```

```
double_t fmax                (double_t x, double_t y);
double_t fmin                (double_t x, double_t y);

#ifdef powerc
long double fmaxl            (long double x, long double y);
long double fminl            (long double x, long double y);
#endif
```

Internal Prototypes

```
long int __fpclassify        (long double x);
long int __fpclassifyd       (double x);
long int __fpclassifyf       (float x);
long int __isnormal          (long double x);
long int __isnormald         (double x);
long int __isnormalf         (float x);
long int __isfinite          (long double x);
long int __isfinited         (double x);
long int __isfinitef         (float x);
long int __isnan             (long double x);
long int __isnand            (double x);
long int __isnanf            (float x);
long int __signbit           (long double x);
long int __signbitd          (double x);
long int __signbitf          (float x);
double __inf                 (void);
```

Non-NCEG Extensions

```
#ifndef  __NOEXTENSIONS__
```

Financial functions

```
double_t compound            (double_t rate, double_t periods);
double_t annuity             (double_t rate, double_t periods);
```

Random Function

```
double_t randomx            (double_t *x);
```

Relational Operator

```
relop relation              (double_t x, double_t y);

#ifdef powerc
relop relationl             (long double x, long double y);
#endif /* powerc */
```

Data Exchange Routines

```
#ifdef powerc
void x80told                (const extended80 *x80, long double *x);
void ldtox80                (const long double *x, extended80 *x80);
#endif /* powerc */
```

Binary-to-Decimal Conversions

```
void num2dec                (const decform *f, double_t x, decimal *d);

#ifdef powerc
void num2decl               (const decform *f, long double x, decimal *d);
#endif /* powerc */

double_t dec2num            (const decimal *d);
void dec2str                (const decform *f, const decimal *d, char *s);
void str2dec                (const char *s, short *ix, decimal *d,
                             short *vp);
#ifdef powerc
long double dec2numl        (const decimal *d);
#endif /* powerc */
float dec2f                 (const decimal *d);
short int dec2s             (const decimal *d);
long int dec2l              (const decimal *d);

#endif      /* __NOEXTENSIONS__ */
#endif
```

Floating-Point Environment Header File (fenv.h)

The file fenv.h defines a collection of functions designed to provide access to the floating-point environment for numerical programming. The file fenv.h declares many functions in support of numerical programming. It provides a set of environmental controls similar to the ones found in the SANE library.

Constants

```
#ifndef  __FENV__
#define  __FENV__
```

Floating-Point Exception Flags

```
#define  FE_INEXACT        0x02000000     /* inexact */
#define  FE_DIVBYZERO      0x04000000     /* divide-by-zero */
#define  FE_UNDERFLOW      0x08000000     /* underflow */
#define  FE_OVERFLOW       0x10000000     /* overflow */
#define  FE_INVALID        0x20000000     /* invalid */

/* The bitwise OR of all exception macros */

#define  FE_ALL_EXCEPT     (  FE_INEXACT | FE_DIVBYZERO | FE_UNDERFLOW | \
                              FE_OVERFLOW | FE_INVALID )
```

Rounding Direction Modes

```
#define  FE_TONEAREST      0x00000000
#define  FE_TOWARDZERO     0x00000001
#define  FE_UPWARD         0x00000002
#define  FE_DOWNWARD       0x00000003

#define  FE_DFL_ENV        &_FE_DFL_ENV   /* pointer to default environment*/
```

Data Types

```
typedef     long int    fenv_t;

typedef     long int    fexcept_t;

/* Definition of pointer to IEEE default environment object */

extern      fenv_t      _FE_DFL_ENV;         /* default environment object */
```

Functions

Controlling the Floating-Point Exceptions

```
void feclearexcept          (int excepts);
void fegetexcept            (fexcept_t *flagp, int excepts);
void feraiseexcept          (int excepts);
void fesetexcept            (const fexcept_t *flagp, int excepts);
int fetestexcept            (int excepts);
```

Controlling the Rounding Direction

```
int fegetround              (void);
int fesetround              (int round);
```

Controlling the Floating-Point Environment

```
void fegetenv               (fenv_t *envp);
int feholdexcept            (fenv_t *envp);
void fesetenv               (const fenv_t *envp);
void feupdateenv            (const fenv_t * envp);

#endif
```

APPENDIX D

FPCE Recommendations for Compilers

This appendix gives some recommendations for what compilers should implement to comply with the FPCE technical report. The PowerPC Numerics library provides much of this compliance, but some aspects of the report must be implemented by the compiler. This appendix describes those features that must be implemented in the compiler and recommends how they should be implemented. You should read this appendix if you are a compiler designer, or if you are a programmer and want to know what numeric features to look for in your compiler.

Environmental Access Switch

To allow compilers to better optimize applications without ignoring the floating-point environment altogether, the FPCE technical report defines the following pragma to be used as an environmental access switch:

```
#pragma fenv_access on | off | default
```

The **environmental access switch** specifies whether an application may access the floating-point environment. Access to the floating-point environment must occur as if at run time, whereas optimizations occur at compile time. At compile time, the default (to nearest) rounding mode is in effect and all exception flags are clear (this is the default environment). Without an environmental access switch, the compiler must always assume that every floating-point expression might produce an exception, and therefore the compiler cannot perform some types of optimizations (such as forward and backward code motion) on floating-point expressions.

If the environmental access switch is supported, whenever programmers use any of the environmental control functions (described in Chapter 8, “Environmental Control Functions”), they should first turn on the switch. Where the switch is on, the compiler does not fully optimize floating-point expressions, because it assumes that that part of the application can access the floating-point environment. (Accessing the floating-point environment means setting the rounding direction or reading the status of the exception flags.) Where the switch is off, the compiler can fully optimize any floating-point expression because it assumes that that part of the application does not access the floating-point environment. If the application accesses the floating-point environment when the switch is off, the result is undefined.

If an application uses the default rounding mode and does not access floating-point exception flags, the programmer may turn off the environmental access switch, allowing the application to be fully optimized. If the application contains modules that must access the floating-point environment, the programmer must turn on the environmental access switch in those modules and turn it off in all other modules. In this way, the modules that do not require access can be fully optimized.

The FPCE technical report recommends these programming conventions:

- A function call must not alter its caller's modes, clear its caller's flags, or depend on the state of its caller's flags unless the function is so documented.
- A function call is assumed to require default modes unless its documentation specifically promises otherwise or unless it does not contain floating-point expressions.
- A function call is assumed to have the potential of raising floating-point exceptions unless its documentation specifically promises otherwise or unless it does not contain floating-point expressions.
- At compile time, the default environment is in effect.

These conventions allow the programmer to ignore the floating-point environment altogether if default modes are sufficient for the application or function.

Where supported, the `fenv_access` pragma can occur only outside external declarations. It enables or disables compiler optimizations until another `fenv_access` pragma is encountered or until the end of the module. The default state for `fenv_access` is implementation dependent.

Contraction Operator Switch

To allow programmer control of whether contraction operators are used, the FPCE technical report defines the following pragma:

```
#pragma fp_contract on | off
```

When the `fp_contract` pragma is turned on, the compiler can produce contraction operators in the generated code. For the PowerPC processor, the contraction operators are the multiply-add instructions. These instructions perform a multiplication operation and either an addition or a subtraction operation with at most a single roundoff error. For some input values, the result of a multiply-add instruction is slightly different than if the operations were performed separately. This difference in value might be unacceptable in certain programs. Compilers that support the `fp_contract` pragma allow programmers to disable the generation of multiply-add instructions where necessary.

Where supported, the `fp_contract` pragma can occur only outside external declarations. It enables or disables contraction operators until another `fp_contract` pragma is encountered or until the end of the module. The default state for `fp_contract` is implementation dependent.

Hexadecimal Floating-Point Constants

The FPCE technical report expands the definition of a floating-point constant in C to include hexadecimal floating-point constants. This format makes it easier to represent constants equal to or near arbitrary powers of 2 because they can be represented in hexadecimal instead of having to be converted to decimal.

A hexadecimal floating-point constant has the form

0x*hex_digit_seq*[.*hex_digit_seq*]p[+|-]*binary_exponent*[*suffix*]

which is interpreted as

$$hex\_digit\_seq.hex\_digit\_seq \times 2^{(+|-)\,binary\_exponent}$$

hex_digit_seq A sequence of hexadecimal digits. The first digit sequence must be preceded by the characters `0X` or `0x`. The hexadecimal point and the digit sequence appearing after it are optional.

binary_exponent
A decimal integer representing a power of 2. The exponent may or may not have a sign, but it must be preceded by the character `p`.

suffix One of the standard C floating-point constant suffixes such as `f` for `float`. All floating-point constants are type `double` unless specified otherwise.

Some examples of hexadecimal floating-point constant expressions are

`0x1.1111p–2    /* interpreted as` $1.1111_{16} \times 2^{-2}$ `*/`
`0x256p35f      /* interpreted as` $256_{16} \times 2^{35}$ `*/`

Implementing an Expression Evaluation Method

Though PowerPC Numerics can recommend certain expression evaluation methods, these methods must be implemented by the compiler. As described in Chapter 3, “Expression Evaluation,” compilers may or may not support widest-need evaluation. This section describes

- the advantages and disadvantages of supporting and not supporting widest-need evaluation

- some special issues compilers must consider regarding evaluating floating-point constants and initializing floating-point variables
- the FPCE-recommended macros and pragmas that help programmers use the most efficient types possible and determine which expression evaluation method is being used

Expression Evaluation Without Widest Need

The main advantage of using an expression evaluation method without widest-need evaluation is that it is simple to implement. The PowerPC architecture is based on single-precision and double-precision operations, so either single or double is a logical choice for the minimum evaluation format.

Choosing single as the minimum format provides the highest performance for single-precision algorithms yet still allows double and double-double algorithms to be performed with greater precision and range. A single minimum evaluation format, then, allows the best possible performance for all expressions by allowing the semantic type of a simple expression to determine its evaluation format.

Choosing double as the minimum format provides extra precision and range to single-precision operations and conforms to the traditional behavior of the C programming language (traditional C performs all floating-point operations in double precision). Performing all single-precision operations in double precision protects the operations against roundoff errors and against encountering an overflow or underflow in an intermediate value. For example, consider the following expression:

$$\frac{10^{38} \times 10^{20}}{10^{20}}$$

If you perform this expression by hand, you get 10^{38}. If all constants are in single format, the expression produces $+\infty$. The constant 10^{38} is near the end of the range of single format. Multiplying by 10^{20} produces 10^{58}, which is rounded to $+\infty$. Then, $+\infty$ is divided by 10^{20}, and the answer is still $+\infty$.

If the minimum evaluation format is double, the constants 10^{38} and 10^{20} are converted to double format before the result is calculated. The multiplication operation no longer overflows the range of the data type because the double format can easily hold 10^{58}. The value 10^{58} divided by 10^{20} produces 10^{38}, which is then converted back to single format.

Choosing the double-double format provides the greatest available precision to all floating-point operations, protecting double-precision operations as well as single-precision operations from roundoff errors. However, it significantly decreases performance for those expressions that would normally be evaluated in a narrower format. In most cases, the extra precision is not necessary.

Imposing a narrow format allows the best possible performance for narrow-format operations but might produce more roundoff errors in places where the extra precision really is necessary. Using widest-need evaluation for complex expressions in conjunction with a minimum evaluation format minimizes the disadvantages of choosing one minimum evaluation format.

Expression Evaluation With Widest Need

Widest-need evaluation provides some of the advantages of using double-double as the minimum format while eliminating the pitfalls. With widest-need evaluation, if an expression contains a double-double variable, all other variables in that expression will ultimately be converted to double-double format, thus reducing the chance of roundoff error in these expressions. If an expression does not contain a double-double variable, widest-need evaluation allows the expression to be evaluated in the narrowest format possible, allowing the best possible performance for that expression.

Widest-need evaluation can seriously inhibit the common subexpression removal optimization for subexpressions of narrower types. If the type of a subexpression is narrower than the type of its enclosing expression, the format of the enclosing expression is imposed on that subexpression. The subexpression's operands are converted to the wider format. Because the conversion must occur as if at run time, the common subexpression removal optimization is in effect disabled for this subexpression.

Floating-Point Constant Evaluation

When a floating-point constant expression appears in a program, the expression evaluation method determines its evaluation format. When widest-need evaluation is not used, the constant is the wider of the minimum evaluation format and the semantic type of the expression. With widest-need evaluation in effect, the constant is converted to the evaluation format of the complex expression it is part of.

In most cases, floating-point constant expressions must be evaluated as if at run time, although they may actually be evaluated at compile time. At compile time, the default rounding direction is in effect, and no floating-point exceptions may be flagged. (These conditions are known as the default floating-point environment. See Chapter 4, "Environmental Controls," for more information.) However, if evaluation takes place as if at run time, the floating-point environment may affect or be affected by the evaluation. This means that if an expression is unexceptional and the default rounding direction is in effect, the expression can be evaluated at compile time. If the expression is exceptional or the current environment is not in the default state, the expression must be evaluated at run time.

In the following two cases the evaluation always takes place at compile time:

- The constant expression appears within the declaration of a variable explicitly declared to be static:

```
static double x = 0.3 + 0.3;
```

- The constant expression appears within the declaration of an aggregate type variable (array, structure, or union):

```
struct {int x = 0; double y = 0.3 + 0.3;} numbers;
```

The requirement that floating-point constant expressions be evaluated as if at run time usually inhibits the constant folding optimization, in which values of constants are combined at compile time to produce fewer operations at run time. However, constant folding can occur

- if a floating-point constant expression is required to be evaluated at compile time (that is, if the expression is part of the declaration of either an explicitly declared static variable or an aggregate type)
- if the evaluation of the expression at compile time has exactly the same results as it would if evaluated at run time. This can happen under the following conditions:
    - If an expression evaluates to be nonexceptional at compile time, it would also evaluate to be nonexceptional at run time.
    - If the expression appears in a portion of the program where access to the floating-point environment is disabled, the default environment will be in effect at run time, just as it is at compile time.

The following example illustrates when floating-point constant expressions are evaluated:

```
#pragma fenv_access on
void f(void) {
   float w[] = {0.0 / 0.0};      /* no exception raised */
   static float x = 0.0 / 0.0;   /* no exception raised */
   float y = 0.0 / 0.0;          /* exception raised */

   x = 1.0 / 4.0;                /* exact (no exception raised) */
   y = 1.0 / 3.0;                /* exception raised */
}
#pragma fenv_access off
void g(void) {
   double z;

   z = 0.0 / 0.0;                /* no exception raised */
}
```

In the declaration of the array `w`, a floating-point constant expression contains division by zero. This operation is evaluated at compile time because it appears in the declaration of an aggregate type. Similarly, the division by zero in the declaration of `x` is evaluated at compile time because it is declared static. Neither of these expressions generates an exception, because they occur at compile time, although the compiler should generate a warning message in each case.

The next declaration (of `float y`) also includes the expression 0.0/0.0. This expression is evaluated at run time, and the invalid-operation exception is raised.

The first statement in function `f` assigns to `x` the value of the floating-point constant expression 1.0/4.0. The compiler looks at this expression to determine if it will raise any exceptions. The expression is found to be exact, so the compiler can optimize it.

The second statement of the function `f` assigns to `y` the value of the floating-point constant expression 1.0/3.0. The compiler determines that this expression will raise the inexact exception, so it must be evaluated at run time. The compiler cannot optimize it.

Finally, function `g` assigns to the double variable `z` the value of the floating-point constant expression 0.0/0.0. This statement appears after the `fenv_access` pragma has been turned off. This pragma (described in the section "Environmental Access Switch" on page D-1) signals to the compiler that the default environment will be in effect at run time. Because exceptions are disabled in the default environment, this statement will not raise a run-time exception, and so it may be evaluated at compile time and optimized.

Initializing Floating-Point Objects

A program achieves better performance if it initializes data (including floating-point data) at compile time. The degree to which this is possible depends on the programming language and the compiler options that are supported.

As specified for the C programming language, floating-point constant expressions are generally evaluated as if at run time. This includes floating-point constants that initialize floating-point variables. However a floating-point variable may be initialized at compile time

- if the variable is declared to be static

  ```
  static float x = 0.3;
  ```

- if the variable is part of an aggregate type

  ```
  struct {int x = 0; float y = 0.3;} numbers;
  ```

- if the initializing value is nonexceptional (exact) and is in the format of the variable

  ```
  double y = 0.0;
  float x = 0.0f;
  ```

- if access to the floating-point environment is disabled in the part of the program where the variable is initialized

  ```
  #pragma fenv_acess off
  float x = 0.3;
  ```

For programming languages other than C, the data initialization model may be simpler. For example, in Fortran static initialization is accomplished with the DATA statement (embedded in a BLOCK DATA subprogram for labeled COMMON initialization), and the initializing values may only be constants or parameters. Such initialization is accomplished as if at compile time. Variables not initialized by the DATA statement are considered uninitialized and are assigned values at execution time with executable statements.

Data initialization rules for Pascal compilers are implementation defined and must be fully documented. In MPW Pascal targeting 680x0-based Macintosh computers, for example, a unit requiring initialization of its data declares a public procedure, called at execution time by the host program, that performs the initialization. Apple II Pascal, on the other hand, supports an initialization section within the unit.

Compiler Extensions for Expression Evaluation

The FPCE technical report recommends that compilers implement two macros that help a programmer determine which expression evaluation method is being used and three pragmas that help a programmer use the most efficient data type for functions.

Determining the Expression Evaluation Method

Two macros that characterize the evaluation method for floating-point expressions may be defined in the float.h header file. The macro _MIN_EVAL_FORMAT tells which numeric data format is used as the minimum evaluation format:

0	float (single)
1	double
2	long double (double-double)

The macro _WIDEST_NEED_EVAL specifies if widest-need evaluation is performed:

0	no
1	yes

Widening for Efficiency

In general, programmers want to use the most efficient floating-point data type for the architecture on which their applications will run. If the application is to run on more than one architecture, you cannot guarantee that the most efficient type on one architecture will be the most efficient type for the others. The FPCE technical report recommends three preprocessor pragmas to facilitate running the same application efficiently on different architectures. When these pragmas are turned on, the compiler uses the wider of the architecture's most efficient type and the declared type for any function, parameter, or local variable declared after the pragma.

```
#pragma fp_wide_function_returns    on | off
#pragma fp_wide_function_parameters on | off
#pragma fp_wide_variables           on | off
```

If the first pragma, fp_wide_function_returns, is turned on in a module, all of the functions defined below the pragma will have return values in the most efficient data type for the architecture if it is wider than the declared return type. If the following example is compiled for the 680x0 architecture, both functions ffunc and ldfunc return type long double. If compiled for the PowerPC architecture, ffunc returns type double and ldfunc returns type long double (because data types may be widened to the most efficient type but not narrowed).

```
#pragma fp_wide_function_returns on
float ffunc (float f) { /* code for ffunc */ }
long double ldfunc (double y) { /* code for ldfunc */ }
```

If the second pragma, fp_wide_function_parameters, is turned on in a module, all of the parameters for all of the functions defined below the pragma are converted to the most efficient data type for the architecture if it is wider than the declared types of the parameters. In the following example, the parameters x and y are both type double on the PowerPC architecture and type long double on the 680x0 architecture. If an architecture's most efficient type was float, the types for both parameters would remain the same (because a parameter's type may be widened to the most efficient type but never narrowed).

```
#pragma fp_wide_function_parameters on
float func(float x, double y) { /* code for func */ }
```

If the third pragma, fp_wide_variables, is turned on in a module, all local variables defined below the pragma are converted to the most efficient data type for the architecture if it is wider than the declared types of the variables. In the following example, the variables z and q are both type double on the PowerPC architecture and type long double on the 680x0 architecture. If an architecture's most efficient type was float, the types for both variables would remain the same (because a variables's type may be widened to the most efficient type but never narrowed).

```
#pragma fp_wide_variables on
float func(float x)
{
   float z;
   double q;

   /* code */
}
```

These pragmas can occur only outside external declarations. Each pragma remains in effect until it is explicitly turned off or until the end of the module. The default state for all three pragmas is off.

If an address or sizeof operator is applied to a widened parameter or variable, a compile-time warning is issued. Casts avoid widening in areas where one of these pragmas is turned on.

APPENDIX E

MathLib Reference

This appendix provides a reference for the numeric implementation in the C programming language. It summarizes the data formats available and tells how to determine the floating-point class for a value. It also lists functions that control the floating-point environment, functions that perform floating-point operations, and the exceptions those functions might raise.

Floating-Point Data Formats

Figure E-1 Floating-point data formats

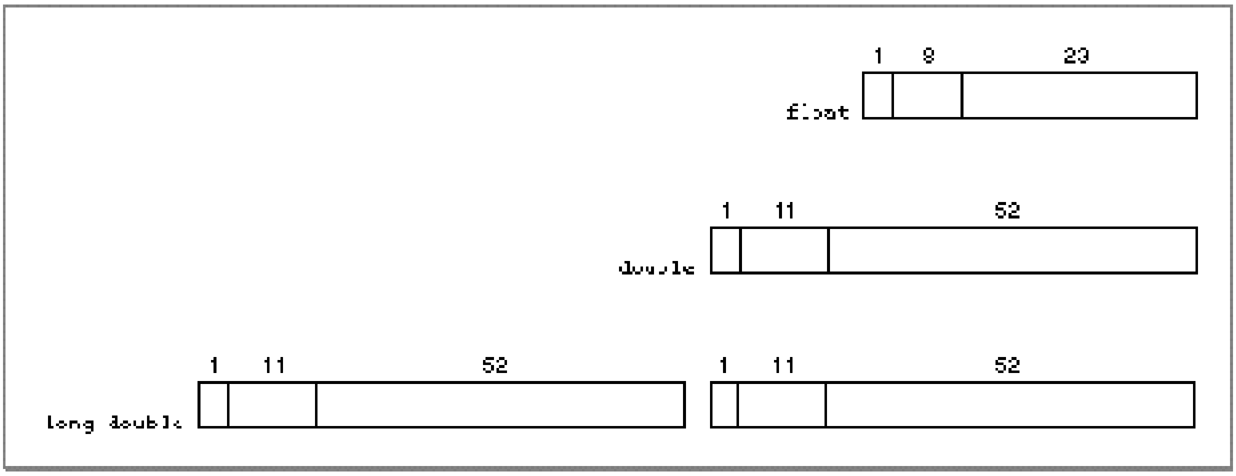


Table E-1 Interpreting floating-point values

If biased[*] exponent *e* is:	And fraction *f* is:	Then value *v* is:	And class of *v* is: [†]
$0 < e < max$ [‡]	(any)	$v = (-1)^s \times 2^{(e - bias)} \times (1.f)$ [§]	FP_NORMAL
$e = 0$	$f \neq 0$	$v = (-1)^s \times 2^{minexp} \times (0.f)$ [¶]	FP_SUBNORMAL
$e = 0$	$f = 0$	$v = (-1)^s \times 0$	FP_ZERO
$e = max$	$f = 0$	$v = (-1)^s \times \infty$	FP_INFINITE
$e = max$	$f \neq 0$	$v = \text{NaN}$	FP_SNAN (first bit is 0) FP_QNAN (first bit is 1)

[*] *bias* = 127 for float, 1023 for double, long double.
[†] From enumerated type NumKind.
[‡] *max* = 255 for float, 2047 for double, long double.
[§] For long double both head and tail are evaluated this way and added together.
[¶] *minexp* = –126 for float, –1022 for double, long double.

Table E-2 Class and sign inquiry macros

```
fpclassify(x)
isnormal(x)
isfinite(x)
isnan(x)
signbit(x)
```

Environmental Controls

Table E-3 Environmental access

Action	Function prototype
Get	`void fegetenv (fenv_t *envp);`
Set	`void fesetenv (const fenv_t *envp);`
Save	`int feholdexcept (fenv_t * envp);`
Restore	`void feupdateenv (const fenv_t *envp);`

Table E-4 Floating-point exceptions

Exceptions	Action	Function prototype
`FE_INEXACT`	Get	`void fegetexcept(fexcept_t *flagp, int excepts);`
`FE_DIVBYZERO`	Set	`void feraiseexcept (int excepts);`
`FE_UNDERFLOW`	Clear	`void feclearexcept (int excepts);`
`FE_OVERFLOW`		`void fesetexcept (const fexcept_t *flagp, int excepts);`
`FE_INVALID`	Test	`int fetestexcept (int excepts);`

Table E-5 Rounding direction modes

Modes	Action	Function prototype
`FE_TONEAREST`	Get	`int fegetround (void);`
`FE_TOWARDZERO`	Set	`int fesetround (int round);`
`FE_UPWARD`		
`FE_DOWNWARD`		

Operations and Functions

Note

Throughout the tables that follow, in the Exceptions column, I = invalid; X = inexact; O = overflow; U = underflow; D = divide by zero. u

Table E-6 Arithmetic operations

Compute	Syntax	Valid input range	Exceptions
Sum	`x + y`	−∞ to +∞	I X O U -
Difference	`x – y`	−∞ to +∞	I X O U -
Product	`x * y`	−∞ to +∞	I X O U -
Quotient	`x / y`	−∞ to +∞	I X O U D
Square root	`sqrt(x)`	0 to +∞	I X - - -

Remainder	`remainder(x,y)`
	`remquo(x,y,quo)`
	`fmod(x,y)`
		−∞ to +∞	I - - - -

Table E-7 Conversions to integer type

Compute	Syntax	Valid input range	Exceptions
Round in current direction	`rinttol(x)`*	-2^{31} to $2^{31}-1$	I X - - -
Add 1/2 to magnitude and chop	`roundtol(x)`*	-2^{31} to $2^{31}-1$	I X - - -

* Return type of `long int`.

Table E-8 Conversions to integer in floating-point type

Compute	Syntax	Valid input range	Exceptions
Round in current direction	`rint(x)`	−∞ to +∞	- X - - -
	`nearbyint(x)`	−∞ to +∞	- - - - -
Round upward	`ceil(x)`	−∞ to +∞	- - - - -
Round downward	`floor(x)`	−∞ to +∞	- - - - -
Add 1/2 to magnitude and chop	`round(x)`	−∞ to +∞	- X - - -
Round toward zero	`trunc(x)`	−∞ to +∞	- - - - -

Table E-9 Conversions between binary and decimal formats

Compute	Syntax	Valid input range	Exceptions
Convert `decimal` struct to binary	`dec2num(&d)`	`decimal` struct	– – – – –
Convert binary to `decimal` struct	`num2dec(&f,x,&d)`	−∞ to +∞	– – – – –

Table E-10 Conversions between decimal formats

Compute	Syntax	Valid input range	Exceptions
Convert `decimal` struct to string	`dec2str(&f,&d,s)`	`decimal` struct	– – – – –
Convert decimal string to struct	`str2dec(s,&ix,&d,&vp)`	Numeric string	– – – – –

Table E-11 Comparison operations

Compute	Syntax	Valid input range	Exceptions
Positive difference or 0	`fdim(x,y)`	−∞ to +∞	- X O U -
Maximum of 2 numbers	`fmax(x,y)`	−∞ to +∞	- - - - -
Minimum of 2 numbers	`fmin(x,y)`	−∞ to +∞	- - - - -
Relationship of x, y	`relation(x,y)`	−∞ to +∞	- - - - -

Table E-12 Sign manipulation functions

Compute	Syntax	Valid input range	Exceptions
Copy the sign	`copysign(x,y)`	−∞ to +∞	- - - - -
$\|x\|$	`fabs(x)`	−∞ to +∞	- - - - -

Table E-13 Exponential functions

Compute	Syntax	Valid input range	Exceptions
e^x	`exp(x)`	−∞ to +∞	- X O U -
2^x	`exp2(x)`	−∞ to +∞	- X O U -
$e^x - 1$	`expm1(x)`	−∞ to +∞	- X O U -
$x \times 2^n$	`ldexp(x,n)`	−∞ to +∞	- X O U -
	`scalb(x,n)`		- X O U -
x^y	`pow(x,y)`	−∞ to +∞	I X O U D

Table E-14 Logarithmic functions

Compute	Syntax	Valid input range	Exceptions
Fraction and exponent fields of floating-point number	`frexp(x,&n)`	−∞ to +∞	`- - - - -`
ln *x*	`log(x)`	0 to +∞	`I X - - D`
$\log_{10} x$	`log10(x)`	0 to +∞	`I X - - D`
ln (*x* + 1)	`log1p(x)`	> –1	`I X - - D`
$\log_{2} x$	`log2(x)`	0 to +∞	`I X - - D`
Exponent field of floating-point number	`logb(x)`	−∞ to +∞	`- - - - D`
Split real number into fractional part and integer part	`modf(x,&y)`	−∞ to +∞	`- - - - -`

Table E-15 Trigonometric functions

Compute	Syntax	Valid input range	Exceptions
cos *x*	`cos(x)`	Any finite number	`I X - - -`
sin *x*	`sin(x)`	Any finite number	`I X - U -`
tan *x*	`tan(x)`	Any finite number	`I X - U -`
arccos *x*	`acos(x)`	–1 to +1	`I X - - -`
arcsin *x*	`asin(x)`	–1 to +1	`I X - U -`
arctan *x*	`atan(x)`	−∞ to +∞	`- X - U -`
arctan *y* / *x*	`atan2(x,y)`	−∞ to +∞	`- X - U -`

Table E-16 Hyperbolic functions

Compute	Syntax	Valid input range	Exceptions
cosh *x*	`cosh(x)`	−∞ to +∞	- X O - -
sinh *x*	`sinh(x)`	−∞ to +∞	- X O U -
tanh *x*	`tanh(x)`	−∞ to +∞	- X - - -
arccosh *x*	`acosh(x)`	1 to +∞	I X - - -
arcsinh *x*	`asinh(x)`	−∞ to +∞	- X - U -
arctanh *x*	`atanh(x)`	–1 to +1	I X - U -

Table E-17 Financial functions

Compute	Syntax	Valid input range	Exceptions
Compound interest	`compound(r,p)`	0 to +∞	I X - - D
Annuity	`annuity(r,p)`	0 to +∞	I X - - D

Table E-18 Error and gamma functions

Compute	Syntax	Valid input range	Exceptions
error	`erf(x)`	−∞ to +∞	- X - U -
1 – error	`erfc(x)`	−∞ to +∞	- X - U -
$\Gamma(x)$	`gamma(x)`	0 to +∞	I X O - -
$\ln(\vert\Gamma(x)\vert)$	`lgamma(x)`	0 to +∞	I X O - -

Table E-19 Miscellaneous functions

Compute	Syntax	Valid input range	Exceptions
Create NaN	`nan(tagp)`	character string	- - - - -
Next representable number after *x* in direction of *y*	`nextafterd(x,y)`	−∞ to +∞	- X O U -
Hypotenuse	`hypot(x,y)`	−∞ to +∞	- X O U -
Random number generator	`randomx(&x)`	1 to $2^{31} - 2$	- - - - -

APPENDIX F

PowerPC Assembly-Language Numerics Reference

This appendix provides a reference for the numeric implementation in PowerPC assembly language. It summarizes the data formats available, how to determine the floating-point class for a value, the FPSCR, instructions that access the FPSCR, and instructions that perform floating-point operations and the exceptions they might raise.

Floating-Point Data Formats

Figure F-1 Floating-point data formats

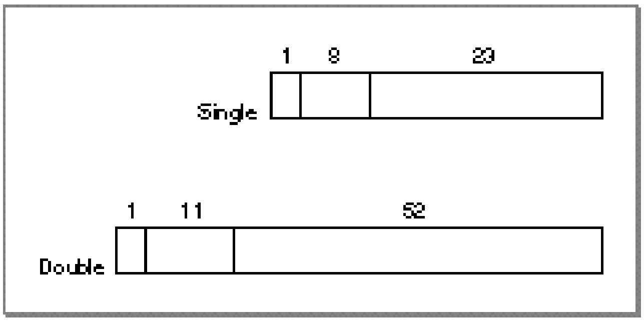


Table F-1 Interpreting floating-point values

If biased[†] exponent *e* is:	And fraction *f* is:	Then value *v* is:	And class of *v* is:
$0 < e < max$[‡]	(any)	$v = (-1)^s \times 2^{(e - bias)} \times (1.f)$	Normalized number
$e = 0$	$f \neq 0$	$v = (-1)^s \times 2^{minexp} \times (0.f)$[§]	Denormalized number
$e = 0$	$f = 0$	$v = (-1)^s \times 0$	Zero
$e = max$	$f = 0$	$v = (-1)^s \times \infty$	Infinity
$e = max$	$f \neq 0$	$v = \mathrm{NaN}$	NaN

[†] *bias* = 127 for single format, 1023 for double format.
[‡] *max* = 255 for single format, 2047 for double format.
[§] *minexp* = –126 for single format, –1022 for double format.

Floating-Point Status and Control Register

Table F-2 Bit assignments for FPSCR fields

FPSCR field	Bit	Meaning if set
0	0	Exception summary
	1	Exception enable summary
	2	Invalid-operation exception summary
	3	Overflow exception
1	4	Underflow exception
	5	Divide-by-zero exception
	6	Inexact exception
	7	Invalid-operation exception; signaling NaN as input
2	8	Invalid-operation exception; $\infty - \infty$
	9	Invalid-operation exception; ∞ / ∞
	10	Invalid-operation exception; $0/0$
	11	Invalid-operation exception; $0 \times \infty$
3	12	Invalid-operation exception; comparison operation
	13	Fraction field rounded
	14	Fraction field inexact
	15	Class descriptor
4	16	< or < 0
	17	> or > 0
	18	= or = 0
	19	Unordered or NaN
5	20	Reserved
	21	Invalid-operation exception; software request (not implemented in MPC601)
	22	Invalid-operation exception; square root (not implemented in MPC601)
	23	Invalid-operation exception; convert-to-integer operation

continued

Table F-2 Bit assignments for FPSCR fields (continued)

FPSCR field	Bit	Meaning if set
6	24	Invalid-operation exception enable / disable
	25	Overflow exception enable / disable
	26	Underflow exception enable / disable
	27	Divide-by-zero exception enable / disable
7	28	Inexact exception enable / disable
	29	Reserved
	30	Rounding direction
	31	Rounding direction

Table F-3 Rounding direction bits in the FPSCR

Modes	Bits	
	30	31
To-nearest	0	0
Upward	1	0
Downward	1	1
Toward-zero	0	1

Table F-4 Class and sign inquiry bits in the FPSCR

Class/sign	Bits				
	15	16	17	18	19
+0	0	0	0	1	0
−0	1	0	0	1	0
Positive normalized number	0	0	1	0	0
Negative normalized number	0	1	0	0	0
Positive denormalized number	1	0	1	0	0
Negative denormalized number	1	1	0	0	0
+∞	0	0	1	0	1
−∞	0	1	0	0	1
Quiet NaN	1	0	0	0	1

Instructions

Note

Throughout the tables that follow, in the Exceptions column, I = invalid; X = inexact; O = overflow; U = underflow; D = divide by zero. In the Instructions column, * = append dot (.) to instruction name to update CR1. u

Table F-5 FPSCR instructions

Instruction	Description	SRC	DST	Exceptions
`mcrfs` *DST*, *SRC*	*DST* ← (*SRC*)	FPSCR field	CR field	- - - - -
`mffs`* *DST*	*DST* ← (FPSCR)	FPSCR	FPR	- - - - -
`mtfsf`* *DST*, *SRC*	*DST* ← *SRC*	FPR	FPSCR field	- - - - -
`mtfsfi`* *DST*, *n*	*DST* ← *n*	16-bit signed int	FPSCR field	- - - - -
`mtfsb1`* *DST*	*DST* ← 1	—	FPSCR bit	- - - - -
`mtfsb0`* *DST*	*DST* ← 0	—	FPSCR bit	- - - - -

Table F-6 Load instructions

Instruction	Description[†]	SRC	DST	Exceptions
`lfd` *DST*, *n*(*GPR*)	*DST* ← (*n* + (*GPR*))	Memory	FPR	- - - - -
`lfdu` *DST*, *n*(*GPR*)	*DST* ← (*n* + (*GPR*)) *GPR* ← *n* + (*GPR*)	Memory	FPR	- - - - -
`lfdx` *DST*, *GPR1*, *GPR2*	*DST* ← ((*GPR1*) + (*GPR2*))	Memory	FPR	- - - - -
`lfdux` *DST*, *GPR1*, *GPR2*	*DST* ← ((*GPR1*) + (*GPR2*)) *GPR1* ← (*GPR1*) + (*GPR2*)	Memory	FPR	- - - - -
`lfs` *DST*, *n*(*GPR*)	*DST* ← (*n* + (*GPR*)) [‡]	Memory	FPR	- - - - -
`lfsu` *DST*, *n*(*GPR*)	*DST* ← (*n* + (*GPR*)) *GPR* ← *n* + (*GPR*)[‡]	Memory	FPR	- - - - -
`lfsx` *DST*, *GPR1*, *GPR2*	*DST* ← ((*GPR1*) + (*GPR2*))[‡]	Memory	FPR	- - - - -
`lfsux` *DST*, *GPR1*, *GPR2*	*DST* ← ((*GPR1*) + (*GPR2*)) *GPR1* ← (*GPR1*) + (*GPR2*) [‡]	Memory	FPR	- - - - -

[†] If *GPR* or *GPR1* is 0, the value 0 is used instead of the contents of the register.
[‡] Converts single to double format automatically.

Table F-7 Store instructions

Instruction	Description[†]	SRC	DST	Exceptions
`stfd` *SRC*, *n*(*GPR*)	*n* + (*GPR*) ← (*SRC*)	FPR	Memory	`- - - - -`
`stfdu` *SRC*, *n*(*GPR*)	*n* + (*GPR*) ← (*SRC*) *GPR* ← *n* + (*GPR*)	FPR	Memory	`- - - - -`
`stfdx` *SRC*, *GPR1*, *GPR2*	(*GPR1*) + (*GPR2*) ← (*SRC*)	FPR	Memory	`- - - - -`
`stfdux` *SRC*, *GPR1*, *GPR2*	(*GPR1*) + (*GPR2*) ← (*SRC*) *GPR1* ← (*GPR1*) + (*GPR2*)	FPR	Memory	`- - - - -`
`stfs` *SRC*, *n*(*GPR*)	*n* + (*GPR*) ← (*SRC*) [‡]	FPR	Memory	`- - - - -`
`stfsu` *SRC*, *n*(*GPR*)	*n* + (*GPR*) ← (*SRC*) *GPR* ← *n* + (*GPR*)[‡]	FPR	Memory	`- - - - -`
`stfsx` *SRC*, *GPR1*, *GPR2*	(*GPR1*) + (*GPR2*) ← (*SRC*) [‡]	FPR	Memory	`- - - - -`
`stfsux` *SRC*, *GPR1*, *GPR2*	(*GPR1*) + (*GPR2*) ← (*SRC*) *GPR1* ← (*GPR1*) + (*GPR2*) [‡]	FPR	Memory	`- - - - -`

[†] If *GPR* or *GPR1* is 0, the value 0 is used instead of the contents of the register.
[‡] Converts double to single automatically.

Table F-8 Conversions to integer format

Instruction	Description	SRC	DST	Exceptions
`fctiw*` *DST*, SRC	*DST* ← (*SRC*) rounded to 32-bit int	FPR	GPR	`I X - - -`
`fctiwz*` *DST*, SRC	*DST* ← (*SRC*) truncated to 32-bit int	FPR	GPR	`I X - - -`

Table F-9 Conversions from double to single format

Instruction	Description	SRC	DST	Exceptions
`frsp*` *DST*, SRC	*DST* ← (*SRC*) rounded to single format	FPR	FPR	`I X O U -`

Table F-10 Comparison instructions

Instruction	Description	SRC	DST	Exceptions
`fcmpo` *DST*, *SRC1*, *SRC2*	*DST* ← (*SRC1*) compare (*SRC2*)	FPRs	CR field	`I - - - -`
`fcmpu` *DST*, *SRC1*, *SRC2*	*DST* ← (*SRC1*) compare (*SRC2*)	FPRs	CR field	`- - - - -`

Table F-11 Arithmetic instructions

Instruction	Description	SRC	DST	Exceptions
`fadd`* *DST*, *SRC1*, *SRC2*	*DST* ← (*SRC1*) + (*SRC2*)	FPRs	FPR	`I X O U -`
`fsub`* *DST*, *SRC1*, *SRC2*	*DST* ← (*SRC1*) – (*SRC2*)	FPRs	FPR	`I X O U -`
`fmul`* *DST*, *SRC1*, *SRC2*	*DST* ← (*SRC1*) × (*SRC2*)	FPRs	FPR	`I X O U -`
`fdiv`* *DST*, *SRC1*, *SRC2*	*DST* ← (*SRC1*) / (*SRC2*)	FPRs	FPR	`I X O U D`

Table F-12 Multiply-add instructions

Instruction	Description	SRC	DST	Exceptions
`fmadd`* *DST*, *SRC1*, *SRC2*, *SRC3*	*DST* ← (*SRC1*) × (*SRC2*) + (*SRC3*)	FPRs	FPR	`I X O U -`
`fmsub`* *DST*, *SRC1*, *SRC2*, *SRC3*	*DST* ← (*SRC1*) × (*SRC2*) – (*SRC3*)	FPRs	FPR	`I X O U -`
`fnmadd`* *DST*, *SRC1*, *SRC2*, *SRC3*	*DST* ← – ((*SRC1*) × (*SRC2*) + (*SRC3*))	FPRs	FPR	`I X O U -`
`fnmsub`* *DST*, *SRC1*, *SRC2*, *SRC3*	*DST* ← – ((*SRC1*) × (*SRC2*) – (*SRC3*))	FPRs	FPR	`I X O U -`

Table F-13 Move instructions

Instruction	Description	SRC	DST	Exceptions
`fabs`* *DST*, *SRC*	*DST* ← |(*SRC*)|	FPR	FPR	`- - - - -`
`fmr`* *DST*, *SRC*	*DST* ← (*SRC*)	FPR	FPR	`- - - - -`
`fneg`* *DST*, *SRC*	*DST* ← – (*SRC*)	FPR	FPR	`- - - - -`
`fnabs`* *DST*, *SRC*	*DST* ← – |(*SRC*)|	FPR	FPR	`- - - - -`

Glossary

680x0-based Macintosh computer Any computer containing a 680x0 central processing unit that runs Macintosh system software. See also **PowerPC processor-based Macintosh computer.**

ANSI X3J11.1 A branch of the American National Standards Institute (ANSI) that is working on a numerics standard for the C programming language. This group is also called the Numerical C Extensions Group (NCEG) and has produced the Floating-Point C Extensions (FPCE) technical report.

antisymmetric Used to describe a function whose graph is not symmetrical across the y-axis; that is $func(x) \neq func(-x)$ for all x.

atomic operations Operations that pass extra information back to their callers by signaling exceptions but that hide internal exceptions, which might be irrelevant or misleading.

bias A number added to the binary exponent of a floating-point number so that the exponent field will always be positive. The bias is subtracted when the floating-point value is evaluated.

binade The collection of numbers that lie between two successive powers of 2.

binary floating-point number A collection of bits representing a sign, an exponent, and a significand. Its numerical value, if any, is the signed product of the significand and 2 raised to the power of the exponent.

complex expression An expression made up of more than one simple expression, that is, an expression with more than one floating-point operation.

Condition Register A 32-bit PowerPC register used to summarize the states of the fixed-point and floating-point processors and to store results of comparison operations.

decimal format structure A data type for specifying the formatting for decimal (base 10) numbers (of conversions). It specifies the decimal number's style and number of digits. It is defined by the `decform` data type.

decimal structure A data type for storing decimal data. It consists of three fields: sign, exponent, and significand (a C string). It is defined by the `decimal` data type.

default environment The environment settings when a PowerPC Numerics implementation starts up: rounding is to nearest and all exception flags are clear.

denormalized number A nonzero binary floating-point number whose significand has an implicit leading bit of 0 and whose exponent is the minimum exponent for the number's data format. Also called *denorm*. See also **normalized number**.

divide-by-zero exception A floating-point exception that occurs when a finite, nonzero number is divided by zero or some other improper operation on zero has occurred.

double format A 64-bit application data format for storing floating-point values of up to 15- or 16-decimal digit precision.

double-double format A 128-bit application data format made up of two double-format numbers. It has the same range as the double format but much greater precision.

environmental access switch A switch, recommended in the FPCE technical report, that specifies whether a program accesses the rounding direction modes and exception flags.

environmental controls The rounding direction modes and the exception flags.

evaluation format The data format used to evaluate the result of an expression. The evaluation format must be at least as wide as the expression's semantic type. (It may be the same as the semantic type.)

exception An error or other special condition detected by the microprocessor in the course of program execution. The floating-point exceptions are invalid, underflow, overflow, divide-by-zero, and inexact.

exception flag Each exception has a flag that can be set, cleared, and tested. It is set when its respective exception occurs and stays set until explicitly cleared.

exponent The part of a binary floating-point number that indicates the power to which 2 is raised in determining the value of the number. The wider the exponent field in a numeric data format, the greater range the format will handle.

expression evaluation method The method by which an evaluation format is determined for an expression.

floating-point operation An operation that is performed on numbers in floating-point formats. The IEEE standard requires that a numerics environment support addition, subtraction, multiplication, division, square root, remainder, and round-to-integer as the basic floating-point arithmetic operations.

Floating-Point Status and Control Register (FPSCR) A 32-bit PowerPC register used to store the floating-point environment.

flush-to-zero system A system that excludes denormalized numbers. Results smaller than the smallest normalized number are rounded to zero.

FPCE technical report A report authored by the Numerical C Extensions Group (ANSI X3J11.1) that proposes a standard for floating-point operations in the C programming language.

FPSCR See **Floating-Point Status and Control Register.**

fraction A field in a floating-point data format that stores all but the leading bit of the significand of a floating-point number.

gradual underflow A computer system that includes denormalized numbers.

IEEE standard A term used in this book to mean IEEE Standard 754.

IEEE Standard 754 A standard that defines how computers should perform binary floating-point arithmetic.

IEEE Standard 854 A standard that defines how computers should perform radix-independent floating-point arithmetic.

inexact exception A floating-point exception that occurs when the exact result of a floating-point operation must be rounded.

Infinity A special value produced when a floating-point operation should produce a mathematical infinity or when a floating-point operation attempts to produce a number greater in magnitude than the largest representable number in a given format. Infinities are signed.

integer types System types for integral values. Integer types typically use 16- or 32-bit two's-complement integers. Integer types are not PowerPC Numerics formats but are available to PowerPC Numerics users.

integral value A value, perhaps in a numeric data format, that is exactly equal to a mathematical integer. For example, –2, –1, 0, 1, 2, and so on.

invalid exception A floating-point exception that occurs if an operand is invalid for the operation being performed.

invalid-operation exception See **invalid exception**.

Machine State Register A 32-bit PowerPC supervisor-level register that records the state of the processor, including if floating-point instructions and floating-point exceptions are enabled.

mantissa See **significand**.

MathLib See **PowerPC Numerics library**.

minimum evaluation format The narrowest format in which a floating-point operation can be performed. Each implementation of PowerPC Numerics defines its own minimum evaluation format.

multiply-add instruction A type of instruction unique to the PowerPC architecture. Multiply-add instructions perform a multiply plus an addition or subtraction operation with at most a single roundoff error.

NaN (Not-a-Number) A special bit pattern produced when a floating-point operation cannot produce a meaningful result (for example, 0/0 produces a NaN). NaNs propagate through arithmetic operations.

NCEG (Numerical C Extensions Group) See **ANSI X3J11.1**.

nextafter functions Functions that return the next value after the input value that is representable in one of the floating-point data formats. For example, nextafterd(0, +∞) returns the value that comes immediately after 0 in the direction of +∞ in double format.

normalized number A binary floating-point number in which all significand bits are significant: that is, the leading bit of the significand is 1. Compare **denormalized number**.

Numerical C Extensions Group (NCEG) See **ANSI X3J11.1**.

overflow exception A floating-point exception that occurs when the magnitude of a floating-point result is greater than the largest finite number that the destination data format can represent.

PowerPC Numerics The floating-point environment on PowerPC processor-based Macintosh computers. This environment provides floating-point data formats and operations plus some advanced numerical functions, such as logarithmic and trigonometric functions.

PowerPC Numerics library A C library that implements floating-point transcendental functions and contains type definitions and macros used for floating-point operations. It is contained in the file MathLib.

PowerPC processor Any member of the family of PowerPC microprocessors. The MPC601 processor is the first PowerPC central processing unit.

PowerPC processor-based Macintosh computer Any computer containing a PowerPC central processing unit that runs Macintosh system software. See also **680x0-based Macintosh computer.**

precision The number of digits required to accurately represent a number. For example, the value 3.2 requires two decimal digits of precision, and the value 3.002 requires four decimal digits. In numeric data formats, the precision is equal to the number of bits (both implicit and explicit) in the significand.

quiet NaN A NaN that propagates through arithmetic operations without signaling an exception.

rounding An action performed when a result of an arithmetic operation cannot be represented exactly in a numeric data format. With rounding, the computer changes the result to a close value that can be represented exactly.

rounding direction modes Modes that specify the direction a computer will round when the result of an arithmetic operation cannot be represented exactly in a numeric data format. Under PowerPC Numerics, the computer resolves rounding decisions in one of the four directions chosen by the user: to nearest (the default), upward, downward, and toward zero.

roundoff error The difference between the exact result of an IEEE arithmetic operation and the result as it is represented in the numeric data format if the result has been rounded.

SANE See **Standard Apple Numerics Environment.**

semantic type The widest type of the operands of an expression.

signaling NaN A NaN that signals an invalid exception when the NaN is an operand of an arithmetic operation. If no halt occurs, a quiet NaN is produced for the result. No PowerPC Numerics operation creates signaling NaNs.

sign bit The bit of a single, double, or double-double number that indicates the number’s sign: 0 indicates a positive number; 1, a negative number.

significand The part of a binary floating-point number that indicates where the number falls between two successive powers of 2. The wider the significand field in a numeric format, the more precision the format has.

simple expression An expression containing one floating-point operation.

single format A 32-bit application data format for storing floating-point values that have a precision of up to seven or eight decimal digits. It is used by engineering applications, among others.

Standard Apple Numerics Environment (SANE) The floating-point environment on 680x0-based Macintosh computers. This environment provides floating-point data formats and operations as well as some advanced numerical functions such as logarithmic and trigonometric functions.

sticky Used to describe a condition in which a bit stays set until it is explicitly cleared. Floating-point exception flags in the FPSCR are sticky, so if one instruction sets an exception flag and another instruction is performed before the flag is tested, it is impossible to tell which instruction caused the exception.

subnormal number A denormalized number.

symmetric Used to describe a function whose graph looks the same on both sides of the y-axis; that is, *func*(*x*) = *func*(–*x*) for all *x*.

tiny Used to describe a number whose magnitude is smaller than the smallest positive normalized number in the format of the number.

transcendental functions Functions that can be used as building blocks in numerical functions. All of the functions contained in the PowerPC Numerics library are transcendental functions.

trigonometric functions Functions that perform trigonometric operations, such as cosine, sine, and tangent.

truncate To chop off the fractional part of a real number so that only the integer part remains. For example, if the real number 1.99999999999 is truncated, the truncated value is 1.

underflow exception An exception that occurs when the result of an operation is both tiny and inexact.

usual arithmetic conversions Automatic conversions performed in the C programming language. The ANSI C specification defines these conversions.

widest-need evaluation An evaluation method in which the widest format of all of the operands in a complex expression is used as the format in which the expression is evaluated.

Bibliography

Aho, A. V., R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.

Alefeld, G., and J. Hertzberger. *Introduction to Interval Computations.* New York: Academic Press, 1983.

American National Standards Institute. *Floating-Point C Extensions*, Prepared by the Floating-Point C Extensions (FPCE) branch of the Numerical C Extensions Group. ANSI X3J11.1 / 93-028, 1993.

Apple Computer. *Apple Numerics Manual*, second edition. Reading, MA: Addison-Wesley, 1988.

Apple Computer. *Inside Macintosh: PowerPC System Software*. Reading, MA: Addison-Wesley, 1994.

Apple Computer. *Assembler for Macintosh With PowerPC.* Cupertino, CA: Apple Computer, 1994.

Apple Computer. *C/C++ Compiler for Macintosh With PowerPC.* Cupertino, CA: Apple Computer, 1994.

Brown, W. S. “A Simple but Realistic Model of Floating-Point Computation.” *ACM Transactions on Mathematical Software* Vol. 7, No. 4 (1981).

Cody, W. J. “Floating-Point Standards—Theory and Practice.” In *Reliability in Computing: The Role of Interval Methods on Scientific Computing*, edited by Ramon E. Moore. Boston, MA: Academic Press, 1988.

Cody, W. J., et al. “A Proposed Radix- and Word-Length-Independent Standard for Floating-Point Arithmetic.” *IEEE Micro* Vol. 4, No. 4 (1984).

Coonen, Jerome T. “An Implementation Guide to a Proposed Standard for Floating-Point Arithmetic.” *IEEE Computer* Vol. 13, No. 1 (1980).

Coonen, Jerome T. “Underflow and the Denormalized Numbers.” *IEEE Computer* Vol. 14, No. 3 (1981).

Coonen, Jerome T. “Contributions to a Proposed Standard for Binary Floating-Point Arithmetic.” Ph.D. Thesis, University of California at Berkeley, 1984. (Available from University Microfilm, Ann Arbor, MI.)

Dekker, T. J. “A Floating-Point Technique for Extending the Available Precision.” *Numerisch Mathematik* Vol. 18, No. 3 (1971).

Demmel, James. “The Effects of Underflow on Numerical Computation.” *SIAM Journal on Scientific and Statistical Computing*, Vol. 5, No. 4 (1984).

Farnum, Charles. “Compiler Support for Floating-Point Computation.” *Software Practices and Experience* Vol. 18, No. 7 (1988).

Fateman, Richard J. “High-Level Language Implications of the Proposed IEEE Floating-Point Standard.” *ACM Transactions on Programming Languages and Systems* Vol. 4, No. 2 (1982).

Floating-Point C Extensions (FPCE) technical report. *See* American National Standards Institute.

Forsythe, G. E., and C. B. Moler. *Computer Solution of Linear Algebraic Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1967.

FPCE technical report. *See* American National Standards Institute.

Goldberg, D. “Computer Arithmetic.” In *Computer Architecture: A Quantitative Approach*, edited by David Patterson and John L. Hennessy. Los Altos, CA: Morgan Kaufmann, 1990.

Golub, G. H., and C. F. Van Loan. *Matrix Computations*. Baltimore, MD: Johns Hopkins University Press, 1989.

Hough, D. “Applications of the Proposed IEEE 754 Standard for Floating-Point Arithmetic.” *IEEE Computer* Vol. 14, No. 3 (1981).

Institute of Electrical and Electronics Engineers. *IEEE Standard for Binary Floating-Point Arithmetic*. IEEE Standard 754-1985. New York: IEEE, 1985.

Institute of Electrical and Electronics Engineers. *IEEE Standard for Radix-Independent Floating-Point Arithmetic*. IEEE Standard 854-1987. New York: IEEE, 1987.

Kahan, W. “Interval Arithmetic Options in the Proposed IEEE Floating-Point Arithmetic Standard.” In *Interval Mathematics* 1980, edited by K. E. L. Nickel. New York: Academic Press, 1980.

Kahan, W. “Rational Arithmetic in Floating-Point.” Berkeley, CA: Report No. PAM-343, Center for Pure and Applied Mathematics, University of California, 1986a.

Kahan, W. “To Solve a Real Cubic Equation.” Berkeley, CA: Report No. PAM-352, Center for Pure and Applied Mathematics, University of California, 1986b.

Kahan, W. “Branch Cuts for Complex Elementary Functions.” In *The State of the Art in Numerical Analysis,* edited by A. Iserles and M. J. D. Powell. New York: Oxford University Press, 1987.

Kahan, W., and Jerome T. Coonen. “The Near Orthogonality of Syntax, Semantics, and Diagnostics in Numerical Programming Environments.” In *The Relationship between Numerical Computation and Programming Languages,* edited by J. K. Reid. New York: North Holland, 1982.

Kulish, U. W., and W. L. Miranker. “The Arithmetic of the Digital Computers: A New Approach.” *SIAM Review* Vol. 28, No. 1 (1986).

Matula, D. W., and P. Kornerup. “Finite Precision Rational Arithmetic: Slash Number Systems.” *IEEE Transactions on Computing* Vol. C-34, No. 1 (1985).

Moore, R. E. *Methods and Applications of Interval Analysis.* Society for Industrial and Applied Mathematics, 1979.

Motorola Corporation. *PowerPC 601 RISC Microprocessor User's Manual*, Motorola Corporation, 1993.

Rice, John R. *Numerical Methods, Software, and Analysis*, second edition. New York: Academic Press, 1992.

Sterbenz, Pat H. *Floating-Point Computation.* Englewood Cliffs, NJ: Prentice-Hall, 1974.

Swartzlander, E. E., and G. Alexopoulos. "The Sign / Logarithm Number System." *IEEE Transactions on Computing* Vol. C-24, No. 12 (1975).

Index

This Apple manual was written, edited, and composed on a desktop publishing system using Apple Macintosh computers and FrameMaker software. Proof pages were created on an Apple LaserWriter Pro printer. Final page negatives were output directly from text files on an Optrotech SPrint 220 imagesetter. Line art was created using Adobe™ Illustrator and Adobe Photoshop. PostScript™, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type is Palatino® and display type is Helvetica®. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier.

WRITER
Jean Ostrem

LEAD WRITER
Tim Monroe

DEVELOPMENTAL EDITOR
Jeanne Woodward

ILLUSTRATOR
Shawn Morningstar

ART DIRECTOR
Betty Gee

PRODUCTION EDITOR
Lorraine Findlay

PROJECT LEADER
Patricia Eastman

COVER DESIGNER
Barbara Smyth

TECHNICAL CONTRIBUTORS
Ali Sazegari, Paul Finlayson, and Kenton Hanson

Special thanks to Scott Fraser, Liz Ghini, Brian Strull, and Allen Watson III.